Techniques for Advanced Android Malware Triage

by

Omid Mirzaei

in partial fulfillment of the requirements for the degree of Doctor in Computer Science and Engineering

Universidad Carlos III de Madrid

Advisor(s):

Dr. José María de Fuentes
Dr. Juan Tapiador

January 2019

I dedicate this dissertation to my unbelievably great family.

My deepest gratitude to my father who has done his best in my home country, Iran, to support me both financially and spiritually from the very beginning of my childhood till now. Without his supports, I could not reach my biggest goal in life. His honesty and hard working at his career have always been an inspiration for me.

My highest respect and deepest gratitude to my mother as well who has spent a huge amount of time and energy to support me emotionally. Her endless enthusiasm and attempt to change my life for the better has always been unbelievable to me. There are no words to express the level of appreciation I do have for her.

My greatest thanks to my kind and awesome brother who has supported me continuously from high-school until today in graduate studies.

A success can hardly achieve in life without having a wonderful and friendly family. Thus, with million thanks to you, I wish you all the best in life. Love you all!!

Acknowledgements

The current thesis is a gesture of research outcomes obtained from my doctoral studies I spent at Universidad Carlos III de Madrid, Spain. Here, I had the opportunity to specialize on a very interesting and demanding area in cybersecurity which was related to the security of smartphone devices in general and the Android operating system in particular.

I would like to thank my supervisors, Dr. Jose Maria de Fuentes and Dr. Juan Tapiador for their supports in both academic and non-academic issues, their valuable advice and their great encouragements during the whole period.

I would like to thank Prof. Arturo Ribagorda, the head of Computer Security Lab (COSEC), for his kind welcome at the beginning when I entered this group, and for his trust on me which gave me the chance to work at this active group and to fulfill the objectives of my doctoral education.

I am willing to thank Dr. Lorena González-Manzano for her kind guidance and encouragements during my doctoral studies.

I am also grateful for the warm welcome, hospitality and technical weekly advice of Dr. Gianluca Stringhini during my research visit from Information Security Research Group at University College London (UCL).

I would like to thank all my friends and PhD students, faculty and former members at COSEC lab, including Roberto, Alejandro, Guillermo, Carmen, Mohammed, Pedro, Ana, José René, Sergio, and Benjamin who helped me a lot during my stay in Madrid.

I am very thankful to the lovely, sociable and peaceful people of Spain who respect all beliefs, religions and races.

Finally, I am grateful for the co-funding of my PhD education that was mainly provided by the Community of Madrid and the European Union.

Published and Submitted Content

The work contained in this dissertation has resulted in 2 journal papers, **one already accepted for publication** and **one currently under review**. Also, it has resulted to **a very competitive conference publication**. Below, all publications are listed along with their details:

- TriFlow: Triaging Android Applications Using Speculative Information Flows,
  PhD candidate role: *first author*,
  Published in: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS'17)*,
  Acceptance Rate: *18% (67/359)*,
  URL: *https://dl.acm.org/citation.cfm?id=3053001*,
  DOI: *10.1145/3052973.3053001*,
  Note: *wholly included in the thesis in Chapter 3*,
  Statement: *Since this source is literally fully included in Chapter 3, for the sake of clarity, I have applied no typographical means to single it out.*

- Androdet: An Adaptive Android Obfuscation Detector,
  PhD candidate role: *first author*,
  Published in: *Journal of Future Generation Computer Systems*,
  Impact Factor: *4.639 (Q1)*,
  URL: *https://www.sciencedirect.com/science/article/pii/S0167739X18309312*,
  DOI: *10.1016/j.future.2018.07.066*,
  Note: *wholly included in the thesis in Chapter 5*,
  Statement: *Since this source is literally fully included in Chapter 5, for the sake of clarity, I have applied no typographical means to single it out.*

- Behavioral Labeling of Android Malware Families,
  PhD candidate role: *first author*,
  Submitted to: *Journal of Expert Systems with Applications*,
  Impact Factor: *3.768 (Q1)*,
  URL: *https://www.journals.elsevier.com/expert-systems-with-applications*,
  Note: *wholly included in the thesis in Chapter 4*,
  Statement: *Since this source is literally fully included in Chapter 4, for the sake of clarity, I have applied no typographical means to single it out.*

Other Research Merits

Other contributions made during this PhD include:

- Privacy models in wireless sensor networks: A survey,
  PhD candidate role: *third author*,
  Published in: *Journal of Sensors*,
  Impact Factor: *2.057 (Q2)*,
  URL: *https://www.hindawi.com/journals/js/2016/4082084*,
  DOI: *10.1155/2016/4082084*
- Dynamic Risk Assessment in IT Environments: A Decision Guide,
  PhD candidate role: *first author*,
  Published in: *Handbook of Research on Information and Cyber Security in the Fourth Industrial Revolution*,
  URL: *https://www.igi-global.com/chapter/dynamic-risk-assessment-in-it-environments/206786*,
  DOI: *10.4018/978-1-5225-4763-1.ch009*

Abstract

Android is the leading operating system in smartphones with a big difference. Statistics show that 88% of all smartphones sold to end users in the second quarter of 2018 were phones with the Android OS. Regardless of the operating systems which are running on smartphones, most of the functionalities of these devices are offered through applications. There are currently over 2 million apps only on the official Google store, known as Google Play. This huge market with billions of users is tempting for attackers to develop and distribute their malicious apps (or malware).

Mobile malware has raised explosively since 2009. Symantec reported an increase of 54% in the new mobile malware variants in 2017 as compared to the previous year. Additionally, more incentive has been provided for profit-driven malware by the growth of black markets. This rise has happened for Android malware as well since only 20% of devices are running the newest major version of Android OS based on Symantec report in 2018. Android continued to be the most targeted platform with the biggest number of attacks in 2015. After that year, attacks against the Android platform slowed for the first time as attackers were faced with improved security architectures though Android is still the main appealing target OS for attackers. Moreover, advanced types of Android malware are found which make use of extensive anit-analysis techniques to evade static or dynamic analysis.

To address the security and privacy concerns of complex Android malware, this dissertation focuses on three main objectives. First of all, we propose a light-weight yet efficient method to identify risky Android applications. Next, we present a precise approach to characterize Android malware based on their malicious behavior. Finally, we propose an adap-

tive learning system to address the security concerns of obfuscation in Android malware.

Identifying potentially dangerous and risky applications is an important step in Android malware analysis. To this end, we develop a triage system to rank applications based on their potential risk. Our approach, called TriFlow, relies on static features which are quick to obtain. TriFlow combines a probabilistic model to predict the existence of information flows with a metric of how significant a flow is in benign and malicious apps. Based on this, TriFlow provides a score for each application that can be used to prioritize analysis. It also provides the analysts with an explanatory report of the associated risk. Our tool can also be used as a complement with computationally expensive static and dynamic analysis tools.

Another important step towards Android malware analysis lies in their accurate characterization. Labeling Android malware is challenging yet crucially important, as it helps to identify upcoming malware samples and threats. A key challenge is that different researchers and anti-virus vendors assign labels using their own criteria, and it is not known to what extent these labels are aligned with the apps' real behavior. Based on this, we propose a new behavioral characterization method for Android apps based on their extracted information flows. As information flows can be used to track why and how apps use specific pieces of information, a flow-based characterization provides a relatively easy-to-interpret summary of the malware sample's behavior.

Not all Android malware are easy to analyze due to advanced and easy-to-apply anti-analysis techniques that are available nowadays. Obfuscation is the most common anti-analysis technique that Android malware use to evade detection. Obfuscation techniques modify an app's source (or machine) code in order to make it more difficult to analyze. This is typically applied to protect intellectual property in benign apps, or to hinder the process of extracting actionable information in the case of malware. Since malware analysis often requires considerable resource investment, detecting the particular obfuscation technique used may contribute to apply the right analysis tools, thus leading to some savings.

Therefore, we propose AndrODet, a mechanism to detect three popular types of obfuscation in Android applications, namely identifier renaming,

string encryption, and control flow obfuscation. AndrODet leverages online learning techniques, thus being suitable for resource-limited environments that need to operate in a continuous manner. We compare our results with a batch learning algorithm using a dataset of 34,962 apps from both malware and benign apps. Experimental results show that online learning approaches are not only able to compete with batch learning methods in terms of accuracy, but they also save significant amount of time and computational resources.

Finally, we present a number of open research directions based on the outcome of this thesis.

Resumen

Android es el sistema operativo líder en teléfonos inteligentes (también denominados con la palabra inglesa *smartphones*), con una gran diferencia con respecto al resto de competidores. Las estadísticas muestran que el 88% de todos los *smartphones* vendidos a usuarios finales en el segundo trimestre de 2018 fueron teléfonos con sistema operativo Android. Independientemente de su sistema operativo, la mayoría de las funcionalidades de estos dispositivos se ofrecen a través de aplicaciones. Actualmente hay más de 2 millones de aplicaciones solo en la tienda oficial de Google, conocida como Google Play. Este enorme mercado con miles de millones de usuarios es tentador para los atacantes, que buscan distribuir sus aplicaciones malintencionadas (o *malware*).

El *malware* para dispositivos móviles ha aumentado de forma exponencial desde 2009. Symantec ha detectado un aumento del 54% en las nuevas variantes de *malware* para dispositivos móviles en 2017 en comparación con el año anterior. Además, el crecimiento del mercado negro (es decir, plataformas no oficiales de descargas de aplicaciones) supone un incentivo para los programas maliciosos con fines lucrativos. Este aumento también ha ocurrido en el malware de Android, aprovechando la circunstancia de que solo el 20% de los dispositivos ejecutan la versión más reciente del sistema operativo Android, de acuerdo con el informe de Symantec en 2018. De hecho, Android ha sido la plataforma que ha centrado los esfuerzos de los atacantes desde 2015, aunque los ataques decayeron ligeramente tras ese año debido a las mejoras de seguridad incorporadas en el sistema operativo. En todo caso, existen formas avanzadas de *malware* para Android que hacen uso de técnicas sofisticadas para evadir el análisis estático o dinámico.

Para abordar los problemas de seguridad y privacidad que causa el *malware* en Android, esta Tesis se centra en tres objetivos principales. En primer lugar, se propone un método ligero y eficiente para identificar aplicaciones de Android que pueden suponer un riesgo. Por otra parte, se presenta un mecanismo para la caracterización del *malware* atendiendo a su comportamiento. Finalmente, se propone un mecanismo basado en aprendizaje adaptativo para la detección de algunos tipos de ofuscación que son empleados habitualmente en las aplicaciones maliciosas.

Identificar aplicaciones potencialmente peligrosas y riesgosas es un paso importante en el análisis de *malware* de Android. Con este fin, en esta Tesis se desarrolla un mecanismo de clasificación (llamado TriFlow) que ordena las aplicaciones según su riesgo potencial. La aproximación se basa en características estáticas que se obtienen rápidamente, siendo de especial interés los *flujos de información*. Un flujo de información existe cuando un cierto dato es recibido o producido mediante una cierta función o llamada al sistema, y atraviesa la lógica de la aplicación hasta que llega a otra función. Así, TriFlow combina un modelo probabilístico para predecir la existencia de un flujo con una métrica de lo habitual que es encontrarlo en aplicaciones benignas y maliciosas. Con ello, TriFlow proporciona una puntuación para cada aplicación que puede utilizarse para priorizar su análisis. Al mismo tiempo, proporciona a los analistas un informe explicativo de las causas que motivan dicha valoración. Así, esta herramienta se puede utilizar como complemento a otras técnicas de análisis estático y dinámico que son mucho más costosas desde el punto de vista computacional.

Otro paso importante hacia el análisis de *malware* de Android radica en caracterizar su comportamiento. Etiquetar el *malware* de Android es un desafío de crucial importancia, ya que ayuda a identificar las próximas muestras y amenazas de malware. Una cuestión relevante es que los diferentes investigadores y proveedores de antivirus asignan etiquetas utilizando sus propios criterios, de modo no se sabe en qué medida estas etiquetas están en línea con el comportamiento real de las aplicaciones. Sobre esta base, en esta Tesis se propone un nuevo método de caracterización de comportamiento para las aplicaciones de Android en función de sus flujos de información. Como dichos flujos se pueden usar para estudiar el uso de cada dato por parte de una aplicación, permiten proporcionar un resumen

relativamente sencillo del comportamiento de una determinada muestra de *malware*.

A pesar de la utilidad de las técnicas de análisis descritas, no todos los programas maliciosos de Android son fáciles de analizar debido al uso de técnicas anti-análisis que están disponibles en la actualidad. Entre ellas, la ofuscación es la técnica más común que se utiliza en el *malware* de Android para evadir la detección. Dicha técnica modifica el código de una aplicación para que sea más difícil de entender y analizar. Esto se suele aplicar para proteger la propiedad intelectual en aplicaciones benignas o para dificultar la obtención de pistas sobre su funcionamiento en el caso del *malware*. Dado que el análisis de malware a menudo requiere una inversión considerable de recursos, detectar la técnica de ofuscación que se ha utilizado en un caso particular puede contribuir a utilizar herramientas de análisis adecuadas, contribuyendo así a un cierto ahorro de recursos. Así, en esta Tesis se propone AndrODet, un mecanismo para detectar tres tipos populares de ofuscación, a saber, el renombrado de identificadores, cifrado de cadenas de texto y la modificación del flujo de control de la aplicación. AndrODet se basa en técnicas de aprendizaje automático en línea (*online machine learning*), por lo que es adecuado para entornos con recursos limitados que necesitan operar de forma continua, sin interrupción. Para medir su eficacia respecto de las técnicas de aprendizaje automático tradicionales, se comparan los resultados con un algoritmo de aprendizaje por lotes (*batch learning*) utilizando un *dataset* de 34.962 aplicaciones de *malware* y benignas. Los resultados experimentales muestran que el enfoque de aprendizaje en línea no solo es capaz de competir con el basado en lotes en términos de precisión, sino que también ahorra una gran cantidad de tiempo y recursos computacionales.

Tras la exposición de las contribuciones anteriormente mencionadas, esta Tesis concluye con la identificación de una serie de líneas abiertas de investigación con el fin de alentar el desarrollo de trabajos futuros en esta dirección.

Biography

Omid Mirzaei is a Ph.D. candidate in the Computer Security Lab (COSEC) at the Department of Computer Science and Engineering of Universidad Carlos III de Madrid (UC3M). His Ph.D. is funded by the Community of Madrid and the European Union through the research project CIBERDINE (Ref. S2013/ICE-3095). His main area of research is computer security. He is particularly interested in mobile security (e.g., security of smartphones and autonomous vehicles), system security, malware analysis and reverse engineering. In addition, he is eager to tackle security issues from a multi-objective perspective, i.e. trying to deal with such problems by consuming the least possible amount of in hand resources. During his Ph.D., he could achieve two outstanding awards, including the third place award from CSAW-Europe'17 and the best previously published paper award from JNIC'18. He has obtained a B.Sc. in Computer Engineering-Software and a M.Sc. in Computer Engineering-Artificial Intelligence from Azad University, Mashhad Branch, Iran. Additionally, he has worked here for three years as a lecturer before joining COSEC at Universidad Carlos III de Madrid.

Contents

List of Figures

List of Tables

1

Introduction

Human life has witnessed a dramatic change with the rapid emergence and evolution of new smart devices in particular, and the Internet of Things (IoT) in general. These devices, either mobile such as smartphones and smart vehicles, or non-mobile like smart TVs and smart buildings, have opened up new opportunities. Nevertheless, growing capabilities of such devices and their wide range of access to private sensitive information pose serious security and privacy risks to their users.

Smartphones are now one of the most widespread devices in daily lives with around 2.53 billion users in 2018 [3]. This world-wide popularity has several reasons. First, they are usually small in size thanks to the improvements in electronics and circuits design. Second, they are light-weight and easily portable. Last but not least, they provide a great range of functionalities based on various built-in equipment and technologies, including high-quality cameras, powerful processors, GPS, cellullar data, virtual intelligent assistants (e.g., Bixby in Samsung or Siri in Apple), and their sensors. Most of these functionalities are currently offered to users through applications which run on different operating systems.

Smartphone operating systems do not have the same market share. Android is by far the dominant operating system at the moment with around 2 billion monthly active users announced by the Google's CEO, Sundar Pichai, in his keynote speech at the Google I/O 2017. Also, its market share is 77.15% as of July 2018 compared to the second most popular operating system, iOS, with a market share of 19.09% [4]. However, this popularity has been abused by malware campaigns and has turned to be the main root of concerns.

Although vulnerabilities have been identified at various layers of Android operating system, applications are known to be the main cause of vulnerabilities for several reasons. Google Play [5], the official market of Android apps, allows everyone to upload any developed applications. Even

though Google has tightened up its security regulations and defense mechanisms, many apps can bypass security checks and enter this repository of apps which is used by millions of users every day. Google Play Protect [6] is another effort made by Google to protect Android smartphone users against malware at real-time. However, despite all these attempts, Android malware can still find their ways into devices and leak users' sensitive information. Furthermore, many tools are available at no cost which help malware developers to reverse engineer famous benign Android apps and to embed their malicious code into a repackaged version of such apps. Ultimately, Android anti-analysis tools (including commercial ones) such as obfuscators and packers are also widely abused by malware authors to hide their malicious intents in Android applications [7] [8].

The rise in Android malware has been explosive since 2009; specially, with the increase in the number of black markets which foster the development of profit-driven malware [9]. Also, new variants of already known malware specimens have outnumbered the new malware families according to various security threat reports [10] [11]. All these show that security enhancing mechanisms and systems are not yet mature enough to reduce the number of Android malware security threats.

Malicious Android applications pose different amounts of risks to users depending on how they treat sensitive user's or device's data. Sensitive data can be in the form of any Personally Identifiable Information (PII) which may be leaked in different ways either through the network or via text messages [12]. Specifically, Android security model is the root cause of many security and privacy risks apps pose to users. For instance, apps can request different permissions whereas they may not use all of the granted permissions (e.g., over-privileged apps). On the contrary, some apps may use third-party libraries which are not inline with their main functionality (e.g., advertising or tracking libraries [13] [14]). As libraries inherit the same set of permissions granted to the apps, they can abuse them without users' consent.

Several tools have been proposed in recent years either to analyze and characterize Android malware or to assess the security and privacy risks posed to users by these malicious applications. Generally, all these tools are classified into two main categories based on the type of features they consider. The first group of tools make use of static features whereas the second group is based on dynamic features extracted during apps' runtime.

The majority of these tools have their own limitations. The first group misses parts of the app's behavior which is exercised at runtime, and, thus, cannot estimate its risk accurately while the second group requires substantial amount of resources both from time and memory.

1.1 Motivation and Objectives

This dissertation is highly motivated by some fundamental issues we found in the area of Android malware analysis.

Firstly, both static and dynamic analysis tools suffer from specific shortcomings. Static analysis tools are usually imprecise and have high false positive rates as they cannot model apps' runtime behavior. Moreover, they cannot scale well with the number of applications. Last but not least, tools which are based on static features can be bypassed at low or no cost using advanced obfuscation techniques. In contrast, dynamic analysis tools have a high runtime overhead considering the fact that may also miss behavior which are not exposed explicitly at runtime. Furthermore, they can be evaded using emulation detection techniques.

Secondly, despite the existence of several Android malware datasets, malware labels (known as families) are not consistent with apps' real behavior [15]. One of the main causes for these inconsistencies is the lack of appropriate standards for naming malware across different vendors [16]. In most cases, these labels are assigned to malicious apps based on static information, including data about the developer, the source country, code structures [17], etc. While these features could be obtained quickly, they might be imprecise as they do not reflect how malware interacts with the victim device and data. Moreover, they can be modified simply to bypass the labeling system [18].

Thirdly, advanced anti-analysis techniques are vastly applied to recent Android malware by both commercial and off-the-shelf anti-analysis tools which hinders their accurate analysis [19]. In particular, obfuscated and packed Android malware are now pervasive [7] [8]. According to a recent study [20], 15% of the apps in the Google Play app store which is supposed to contain benign apps are obfuscated. This amount is much higher in the malicious apps that are present in the wild. Furthermore, both benign and malicious apps make use of different packing techniques to hide their code.

The complexity and diversity of tools and techniques for either obfuscation or packing have introduced new barriers in protecting Android users.

The main goal of this dissertation is to address all the above issues we encountered during our studies on Android malware analysis. Thus, we have set the following main objectives:

- Developing a fast yet accurate triage system to identify Android applications with potentially dangerous and risky behaviors.
- Proposing a new characterization and labeling scheme for Android malware which does not only rely on static features.
- Alleviating the difficulties in the current analysis of advanced Android malware which are obfuscated with popular obfuscation techniques.

1.2 Contributions and Organization

In this PhD dissertation, we have made the following contributions:

- A fast triage system, known as TRIFLOW, is presented in this thesis. It is a lightweight information flow based triage mechanism to identify Android apps with potentially dangerous behavior. Here, we do rely on information flows which can be effectively used to give an informative summary of an application's behavior. TRIFLOW introduces the notion of speculative information flows. This means that TRIFLOW extracts some features from apps, and, then, predicts the existence of a flow based on them. Each predicted flow is then scored by TRIFLOW in terms of its potential risk, which depends on the flow's observed prevalence in malware and benign applications.
- A behavioral-based labeling for Android malware is proposed in this thesis based on information flows. In particular, our scheme to characterize malware is based on patterns of flows rather than flows themselves. Although flows are interesting as they show how apps treat sensitive data, they provide limited information about apps' behavior. Therefore, our approach considers a larger scale which is different combinations of flows which do appear together in each application.

- An adaptive Android obfuscation detection system, called AndrODet, is developed and proposed in this thesis with some outstanding features. first, it is a modular system to detect three common types of obfuscation in Android applications. Second, it is to deal with multidex Android applications. Last but not least, it is an adaptive system which can improve its accuracy over time and does not need to be re-trained.

The organization of this dissertation is as follows. We first provide the readers with some background information in Chapter 2. Chapter 3 presents our fast triage system to identify risky Android applications. Behavioral-based labeling of Android malware is discussed in Chapter 4 followed by an adaptive system to detect three common obfuscation techniques in Android applications in Chapter 5. We finally conclude this dissertation and propose some future research directions in Chapter 6.

2

Background

This chapter contains the background needed for readers to understand the whole proposals of this dissertation. In particular, we explain the Android's architecture and its security model in Section 2.1. Then, Dalvik bytecode is briefly presented in Section 2.2 followed by an overview of the Android application's structure in Section 2.3. Section 2.4 discusses the evolution of Android malware from the very beginning till today, and, also, provides the readers with some predictions about future types of Android malware that might come into existence. Android malware research datasets are all presented in Section 2.5, and some common techniques used for Android malware analysis are provided in Section 2.6. Furthermore, anti-analysis techniques used by Android malware to evade detection are briefly explained in Section 2.7, and we end this chapter with a discussion of data mining tools for Android malware analysis in Section 2.8.

2.1 Android's Architecture and Security Model

This section presents the Android's architecture and some basic concepts related to its security model.

2.1.1 Android's Architecture

Android operating system consists of several layers which are usually composed of different components written either in Java or other programming languages such as C/C++. Some of these components are developed by third-party contributors of the Android Open Source Project (AOSP) such as the Original Equipment Manufacturers (OEM) [21].

A Linux kernel forms the basis of Android operating system architecture as shown in Fig. 2.1. Similar to other Unix-based systems, this kernel provides all required drivers ranging from hardware and filesystem access

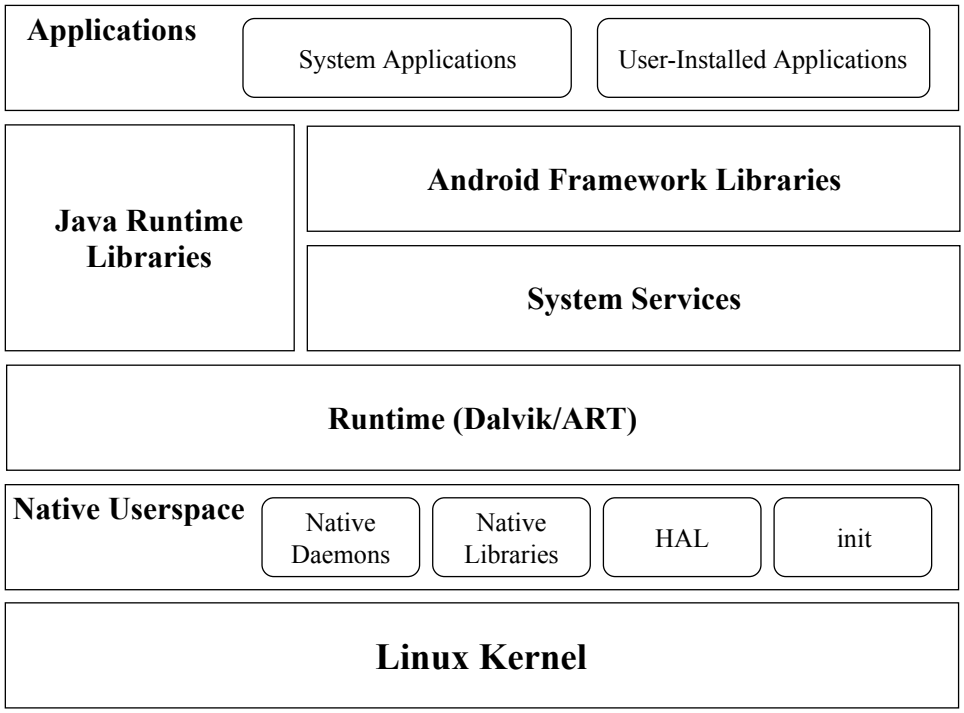


Figure 2.1: The Android OS architecture [1].

to process management and networking. However, it is different from the Linux kernels you may find in other devices or operating systems. Particularly, the difference is in the new features (known as Androidisms [22]) added to the Linux kernel of Android from which Binder and paranoid networking are the most important ones. The most important role of Binder is to handle the Inter-Process Communication (IPC) between applications, whereas paranoid networking manages applications' access to network sockets.

On top of the Linux kernel, there is an important layer, called Native Userspace, that consists of four main components, including Native Daemon, Native Libraries, Hardware Abstraction Layer (HAL) and init. The binary file, init, is the first process that starts running, and, next, starts all other processes. Also, several native libraries (Non-Java libraries which are mainly used by the system like C or C++ libraries) and native daemons (programs written in languages other than Java and are running in the background) are accessible at this layer. The HAL component is a bridge between high-level representations of the hardware used in the libraries and low-level representations used by the kernel. It defines a standard interface for hardware vendors to implement that enables Android to be agnostic about lower level driver implementations.

As Android apps are developed mainly in Java, they need to be compiled and then executed by a Java Virtual Machine (JVM) which is known as Runtime in the Android architecture. Dalvik was the first runtime introduced in the earlier versions of Android (before Android 4.4 or KitKat) which was based on Just-In-Time compilation (JIT). Thus, apps' source codes were compiled and then executed dynamically at runtime. However, due to performance reasons, this runtime was replaced by another variant, called Android Runtime (ART). ART is based on Ahead-Of-Time (AOT) compilation, and, thus, apps are compiled using the on-device dex2oat tool at install time. This can lead to a significant saving in memory consumption.

Runtime Java libraries, defined mainly in java.* and javax.* packages, are provided by the Java Runtime Libraries. These libraries may have some native code dependencies as well. These native codes are linked into Android's core Java libraries by Java Native Interface (JNI). JNI handles both calls to native codes from Java codes and the ones to Java codes from native codes. This layer is directly accessible from the Applications layer as well as system services.

System Services implement the most fundamental features of Android operating system, including telephony, network connectivity and display. Each system service defines a remote interface that can be called from other applications and services. Similar to runtime Java libraries, services are mainly implemented in Java though some others are written in native code.

Android Framework Libraries (also called framework) are in the next layer of Android architecture. The framework includes all Java libraries which are for the most part hosted under the *android* top-level package and are not part of the standard Java runtime (java.*, javax.*, etc.). It also includes the basic classes needed to develop Android applications such as the ones for creating activities, services, content providers (in the android.app.* packages) and GUI widgets (in the android.view.* and android.widget packages) to name a few. Furthermore, it provides necessary classes to app developers in order to interact with device hardware.

Applications stand on the topmost layer of Android operating system architecture. Apps are programs with which smartphone users do interact directly. There exists two common types of apps, known as system appli-

cations and user-installed applications. The apps in both groups have the same structure and are built on top of Android framework.

System apps (typically mounted as */system*) are included in the Android operating system's image, and, thus, cannot be manipulated or uninstalled by users. However, users can update these apps if and only if they are signed with the same private key. While these apps were treated similarly in the earlier versions of Android (up to Android Jelly Bean) by giving them the same number of permissions to access critical resources, only privileged system apps (*installed in /system/priv-app/*) are allowed to access these sensitive resources in the new versions of Android (from Android KitKat).

User-installed apps (typically mounted as */data*) can be uninstalled at any time by users. These apps are encapsulated in their own security sandboxes and cannot affect apps in different sandboxes, neither can they access other apps' data. Moreover, user-installed apps are only allowed to access resources for which they have been granted a permission.

2.1.2 Android's Security Model

Similar to its architecture, Android's security model takes advantage of some security features offered by the Linux kernel. In what follows, primary security features of Android operating system are discussed briefly.

2.1.2.1 Application Sandboxing

One of the most important security features of Android is application sandboxing, also known as application isolation. In a Linux system, users' resources as well as processes are isolated since it is a multi-user operating system. Thus, a user cannot access other users' files without having an explicit permission from the operating system. Here, each process is assigned a unique identity (called User ID or UID) based on the user that has started it.

Android follows the same strategy in isolation but for applications rather than users since smartphones are commonly known as personal devices. To do so, it assigns a unique UID (or app ID) to each application upon installation. It then executes each application in a dedicated process (process-level isolation) with that UID. A directory is also considered for each app that other apps cannot read or write without required permissions (file-level

isolation). This isolation at both process-level and file-level is called application sandboxing. This kernel-level isolation is applied to all apps which are executed either in a native or a virtual machine process. It is worth mentioning that some apps (most often system applications) can have the same UID, also known as shared ID. In this case, they may share some system files and would run in one single process.

2.1.2.2 Permissions

As Android apps are isolated in sandboxes, they can solely access their own files and any publicly available resources on the device. However, apps cannot provide users with a rich functionality unless they are allowed to access some additional resources. Such an extra access right is known as permission. Android can control the access to many sensitive resources such as sensors' data and Internet connectivity through permissions that can be enforced at various levels.

Apps should declare and request the required permissions in their manifest file (discussed in Section 2.3). Permissions granted to the applications during installation can be revoked or re-granted again at runtime in the recent versions of Android (starting from Android Marshmallow). In the earlier versions however, permissions could not be revoked once they were granted to the apps. User-installed apps can define custom permissions as well, restricting the access to an app's resources and services to only those apps which are created by the same developer.

2.1.2.3 Inter-Process Communication

Similar to other Unix-like operating systems, a process in Android cannot access the memories of other processes and have its own address space. In other words, process isolation is another security feature Android inherits from its Linux kernel akin to Unix OS. Isolating processes improves the stability and security of the apps which are running on the device. However, there are some cases in which one process may need to provide useful service(s) for other processes.

Inter-Process Communication (IPC) is a mechanism in Android that allows processes to discover and interact with services that are offered by

other processes. Due to the reliability and flexibility issues with the standard IPC mechanisms, a customized version, called Binder, has been developed for Android to handle communications between processes. Binder uses a combination of userspace libraries and kernel driver to implement IPC effectively.

2.1.2.4 SELinux

Another important difference between the Android's Linux kernel and other Linux kernels (see 2.1.1 for others) is in the way they are implemented. The Android's security model relies on apps isolation through creating sandboxes. While this prevents unauthorized access to each app's files by other apps, it can still grant world access to its files either intentionally or because of programming errors. Inappropriate permissions assigned to an app or system files are shown to be the main source of vulnerabilities in Android. Thus, the access control model of standard Linux kernel has been modified and re-implemented in Android.

In the standard Linux kernel access controls follow a discretionary mode, known as Discretionary Access Control (DAC). This implies that when users are granted necessary permissions to access particular resources, they can pass the same permissions and access levels to other users. In a different access control model, called Mandatory Access Control (MAC), users' access to resources conforms to a set of policies which can only be changed by an administrator, and, thus, users are not allowed to pass the acquired privileges to others. The Android's Linux kernel has been implemented using MAC access control mechanism and is known as Security Enhanced Linux (SELinux). It isolates core system daemons and user-installed applications in different security domains and defines different access policies for each domain.

2.1.2.5 Code Signing

All Android applications, including system and user-installed apps, must be signed by their developers (see 2.3). As these apps are written mostly in Java, the same JAR signing method can be used to sign Android apps. One important advantage of code signing is that operating system can check if user-installed apps are coming from the same sources upon their updates

by comparing their signing certificates. This is known as the same origin policy in Android.

System apps are also signed by a number of platform keys. These keys are generated and controlled by whoever maintains the Android version installed on a particular device, including device manufacturers, carriers or even users for self-developed open source Android versions. System apps which are signed by a common platform key can share resources and run inside the same process.

2.1.2.6 Multi-User Support

Android did not support multiple users in the earlier versions as it was basically developed for smartphones that had a single physical user (see 2.1.2.1). However, it supports multiple physical users starting from Android Jelly Bean (version 4.2) and is only enabled on tablets which are assumed to be used by more than one user.

To support multiple users, Android assigns a unique UID and data directory (or system directory under */data/system/users/<UID>/*) to each user. Then, to distinguish apps for each user, it assigns a new effective UID to each application when it is installed for a particular user. The effective UID is obtained from the target physical user's UID and the app's UID in a single-user system (the app ID). This way of managing users and apps guarantees that applications will be limited to their own sandbox even if multiple instances of those apps are installed by multiple users.

2.2 Dalvik Bytecode

Android programs are written mostly in Java, although they can contain calls to binaries and other shared libraries known as native components [23]. Once written, they are compiled to Java bytecode and, then, to Dalvik bytecode. The final result is a Dalvik EXecutable (DEX) file with a .dex format or an optimized version of it with an .odex format.

The Dalvik Virtual Machine (DVM) is a register-based machine which executes Dalvik bytecode instructions (through a shared library, called libdvm.so) and provides a Java-level abstraction for the Java components of applications [24], while Java Native Interface (JNI) supports the use of native components. DVM is based on Just-in-Time (JIT) compilation and

is replaced by Android RunTime (ART) after Android version 4.4, which works based on Ahead-Of-Time (AOT) compilation and has led to significant improvements in performance and memory consumption [25].

Analyzing Dalvik bytecode is simpler than machine code, it has a better readability for human analysts, and it provides better semantic information. Also, it is easy to be reverse engineered using tools like Dexdump [26], Dex2jar [27], Androguard [28], and Apktool [29] to name a few. Thus, many static malware analysis tools [9], deobfuscators [7] [20], and unpackers [30] [31] have been proposed which extract their features directly from Dalvik bytecode. For instance, key program features such as method names, class names, field names, variables, and strings are very quick to obtain from the .dex file and give useful preliminary information.

2.3 Android Application Structure

Android applications which are installed on the device are in the form of application package files (or APK files) with .apk extensions [1]. Indeed, they are compressed (or ZIP files) that contain both the apps' source codes, their resources and other relevant information. In particular, a typical APK file is consisted of a manifest file (explicitly named as *AndroidManifest.xml*), a dex file (*classes.dex*), a resource file (*resources.arsc*) and four main directories, including *assets*, *lib*, *META-INF* and *res* (Fig. 2.2).

The manifest file contains some fundamental information for each Android application, including its package name, its components, list of permissions which may be requested to access protected system or app resources, and, last but not least, hardware and software features each app requires. An application can have up to four main components, namely activities, services, broadcast receivers and content providers. Activities are single and focused things each user can do. Thus, they are in form of a single screen with a user interface. Services are components that do not have any user interfaces and run in the background. Broadcast receivers are responsible for reacting to system-wide events called broadcasts, and content providers provide interfaces for apps' data that are typically stored in databases.

The dex file is an executable code of each application. Android programs are written in Java programming language though they can have calls to binaries and other shared libraries known as native components

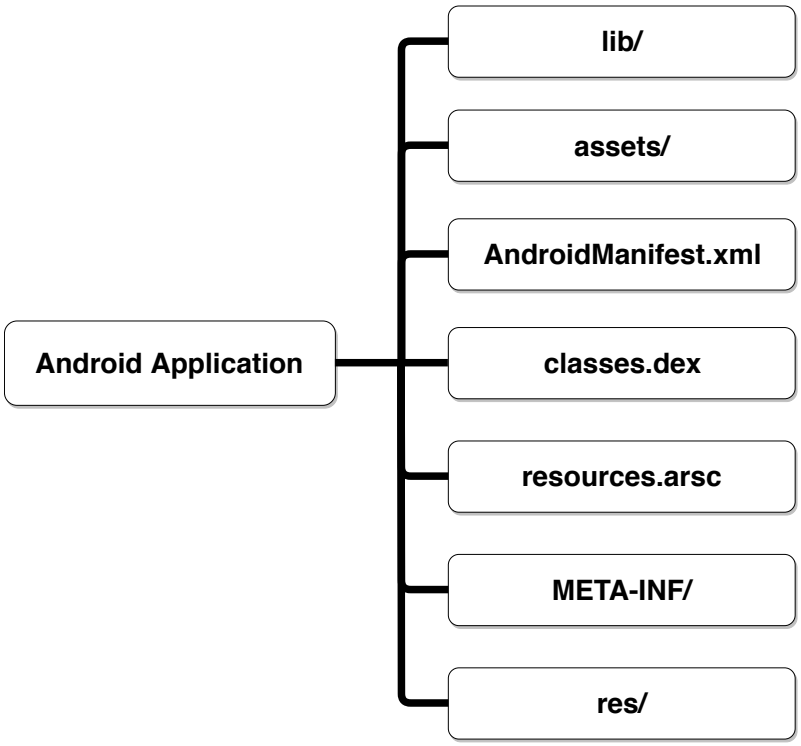


Figure 2.2: The structure of a typical Android application.

[23]. Once written, they are compiled to Java bytecode, and, then, to Dalvik bytecode. The final result is a Dalvik EXecutable (DEX) file in .dex format or an optimized version of it in .odex format.

The resource file packages all application's compiled resources such as strings and styles which are stored in separate binary XML files. Three types of resources are identifiable for an application, including colors, strings and dimensions. Strings usually represent labels that are visible in the user interface of the application while dimensions and colors show sizes and colors of such interfaces respectively. Note that each resource may have several configurations. Thus, the appropriate value is chosen and assigned to a resource at runtime by the Android runtime based on the configuration of the environment it is running on.

The *assets* directory bundles raw asset files with each application. Sounds, user interface pictures and fonts are some files that can be typically found in this directory. Applications can take advantage of native libraries through Java Native Interface (JNI). These apps do contain an extra directory called *lib* with sub-directories for each supported platform architecture.

Android applications' resources, referenced either directly using the an-

droid.content.res.Resources class or indirectly via higher-level APIs, are stored in *res* directory. However, each type of resource (e.g., animation, menu, image, etc.) is stored in a separate sub-directory. The *META-INF* directory contains app's package manifest file and code signatures similar to JAR files. Android controls which apps are authorized to get permissions with the signature protection level that is performed through APK code signing.

Java code signing is performed by reusing and extending JAR manifest files in order to add a code signature to the JAR archive. Each Android application has a main JAR manifest file (*MANIFEST.MF*) whose entries are app's filenames and their corresponding digest values. Also, another manifest file (with .SF extension such as *CERT.SF*), known as signature file, is used in the process of signing which contains the data to be signed and a digital signature. This signature is stored in the same directory as a binary file with one of the .RSA, .DSA, or .EC extensions, depending on the adopted signature algorithm (*CERT.RSA*). Android apps are commonly signed and verified by the official JDK tools for JAR signing and verification, jarsigner and keytool commands [32].

2.4 Android Malware Evolution

The evolution of Android malware may be anticipated to be quite similar to desktop malware initially. However, Android's great market share as well as its openness and its unique security features have all provided an interesting opportunity for the attackers. In addition, smartphones have introduced new technologies that can lead to more diverse and bigger attack vectors. For instance, mobile cellular data, high-quality cameras and sensors and location-aware services are appealing launching points for attackers [9]. All these reasons have made Android malware evolve faster over time [33].

In this section, we provide the readers with an evolution timeline of Android malware (Fig. 2.3) and some relevant information specific to each year gathered from security threat reports of popular cyber security firms, including F-Secure (Finish company), Kaspersky Lab (Russian company), McAfee (American company), Symantec (American Company), Trend Micro (Japanese company), and, finally, Avira (German company).

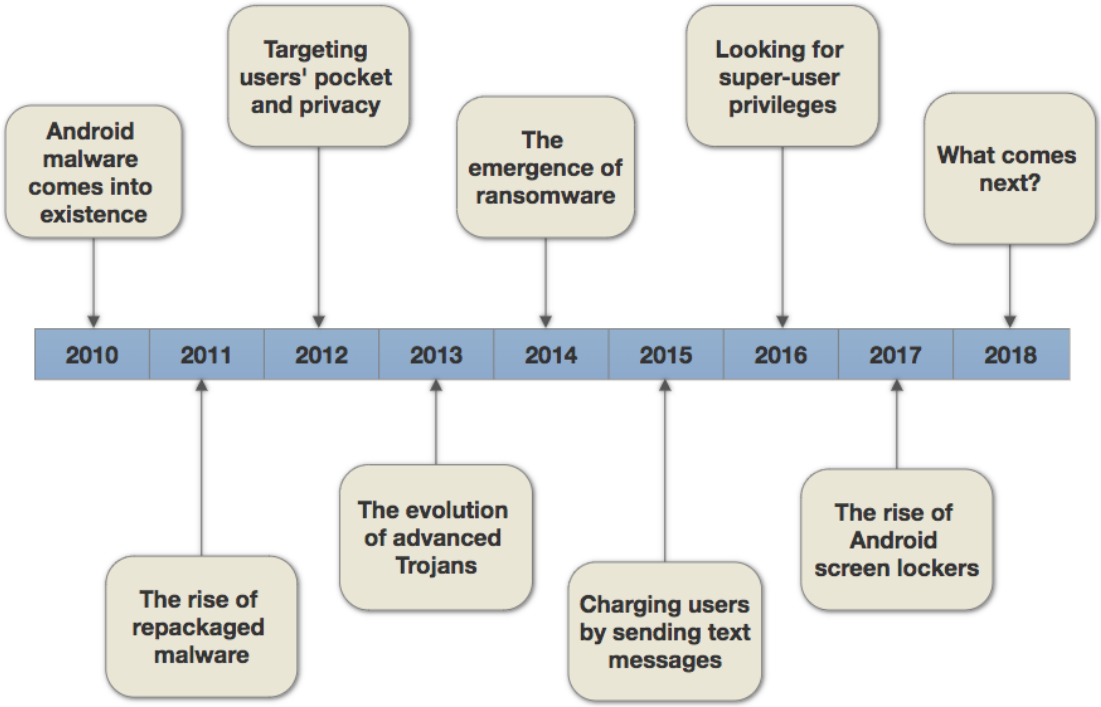


Figure 2.3: The Android malware evolution over years.

2.4.1 2010 - Android Malware Comes into Existence

The first malware in Android operating system was first detected in August 2010 [34]. This Trojan family, known as FakePlayer, pretended to be a media player for watching porn video clips in Android [35]. However, it was just an app that sent SMS text messages (with 798657 as body) to premium-rate numbers (in this case, 3353 and 3354 numbers) without user's consent (Fig. 2.4). It had an icon in its *res* directory (recall Section 2.3) which looked very similar to the legitimate Windows Media Player application.

FakePlayer was an easy-to-analyze malware family with only 3 main classes and almost 350 lines of code. It was specifically developed to target Russian smartphone users for two main reasons. First, Android official market (known as Android Market in 2010) was not accessible world-wide, and, second, money could be sent easily on Eastern European telecom carriers.

Later on, cyber security companies discovered the first spyware in Android malware, called Tapsnake [36] [37]. Tapsnake was a Trojan embedded into a game (Tap Snake) capable of leaking location information

```
.line 35
invoke-static {}, Landroid/telephony/SmsManager;->getDefault()Landroid/telephony/SmsManager;
move-result-object v0
.line 54
.local v0, "m":Landroid/telephony/SmsManager;
const-string v1, "3353"
.line 55
.local v1, "destination":Ljava/lang/String;
const-string v3, "798657"
.line 57
.local v3, "text":Ljava/lang/String;
const/4 v2, 0x0
const/4 v4, 0x0
const/4 v5, 0x0
```

Figure 2.4: An excerpt of smali code extracted from one of the main FakePlayer classes.

to a remote server every 15 minutes. It could also allow a third-party to track user's geographical location by installing another paid app, called GPS Spy. Two key registration information, including an email address and a key were required to be typed in both Tapsnake and GPS Spy to enable geo-location tracking. These information were acquired by social engineering techniques.

Geinimi was another newsworthy Android malware family initially detected by Lookout at the end of 2010 [38]. This family was the most sophisticated malware found by that time with innovative capabilities. Firstly, it was a repackaged version of legitimate Android apps like President vs. Aliens (an educational app) and Baseball Superstars 2010 (a game app) to name a few. Secondly, it made use of two anti-analysis techniques to hide its malicious behavior, including network communication encryption and network communication code obfuscation. Thirdly, it had some botnet-like capabilities with more than 20 commands implemented in its source code. Since then, Geinimi has been found in a wide range of locations, ranging from unofficial marketplaces to official ones.

2.4.2 2011 - The Rise of Repackaged Malware

Malicious apps increased in number from 84 samples in 2010 to more than 1000 in 2011 [39] [40]. Also, inspired by the success of Geinimi in 2010, repackaging benign apps was rapidly imitated by malicious apps from several new families such as ADRD, Pjapps, DroidDream, DroidDreamLight, Zsone, BaseBridge, DroidKungFu, jSMSHider, GoldDream and AnserverBot [35] in 2011. In total, around 86% of malware specimens found in 2010 and 2011 were repackaged versions of legitimate apps [39].

ADRD and Pjapps were the first Trojan families detected in February 2011. Both of these families could steal sensitive device information, including International Mobile Equipment Identity (IMEI), International Mobile Subscriber Identity (IMSI), device ID, line number, subscriber ID and SIM serial number once special conditions were met. Later, they sent this information to remote servers. These types of malware are commonly known as Logic Bomb Trojans. ADRD could be activated once the operating system started, the network connectivity changed or whenever the device received a phone call. However, Pjapps only started execution when the device signal changed. In addition to stealing capability, ADRD family could change the device settings to enable/disable data connectivity in order to send or receive information all of which would result in high data usage. Also, Pjapps was able to send text messages, install a new application and to add a browser bookmark.

DroidDream and DroidDreamLight were the next family of malware discovered one and three years later which repackaged legitimate Android applications to compromise the target device. Similar to the earlier detected families in this year, they could steal sensitive information, including IMEI, IMSI, device model, device language and country and leak them to remote servers. DroidDreamLight could even steal information from text messages, contact lists, call logs and Google account credentials. The main difference between these two families was in the way they behaved in the infected device. DroidDream tried to get root privileges at the first step, after which It was able to manipulate the device settings, system files and SD card. On the contrary, DroidDreamLight ran in the background silently and leaked the sensitive information which were stored encrypted in a configuration file.

Zsone was another malware family found exactly at the same month as DroidDreamLight though with quite different characteristics. This Trojan was developed to target Chinese users mainly. Once installed on the device, it could send text messages to premium-rate numbers related to subscription for SMS-based services to charge the user. It also kept a record of its sending status in an XML file. Incoming text messages from certain numbers could also be intercepted by this family.

BaseBridge and DroidKungFu - both with several variants and stealing capabilities - were the next families of Android malware detected in 2011. Additionally, similar to DroidDream family, they first attempted to gain

root privileges in the target device [41]. Once successful, they stole a variety of information, including IMEI, device model and manufacturer and OS version and sent them to remote servers. BaseBridge was also capable of sending/removing text messages, dialing phone numbers and monitoring phone usage. On the other hand, DroidKungFu could install/remove any packages and modify the homepage of web browser without user's consent.

jSMSHider was the second malware family with big differences in behavior comparing with the earlier discovered families. It was a Trojan which affected smartphones with a customized ROM. It could exploit a vulnerability in these devices and install a payload onto the ROM to communicate with a remote server and receive operational commands. It was then able to read, send and remove text messages (to hide its malicious intent), install apps on ROMs, perform a silent install or update of the APK and download an application from a URL.

Detected after jSMSHider, GoldDream was another logic bomb Trojan with versatile stealing capabilities. It could record the date, time, message body and the senders of text messages upon receiving messages. It could also store the date, time and phone number once any calls were made or received. In addition, it was able to make phone calls, send text messages and install packages without user's knowledge. Device ID, SIM serial number and Subscriber ID were among the most important sensitive information this malware family could steal from the compromised device.

AnserverBot was the third family with different behavior in contrast to its predecessors. It first tried to lure the device user by showing fake upgrade dialog boxes for the host app. It then installed the first payload that could run silently in the background. Afterwards, a second payload could be loaded and executed dynamically either by the host app or the first payload. Finally, it could connect to the remote server to take the commands. This family was one of the pioneers in using anti-analysis techniques as well. First of all, most of the code in both payloads were obfuscated. Second, AnserverBot had a mechanism to avoid the infected application from being repackaged again. Last but not least, it could detect the existence of three smartphone anti-virus software.

2.4.3 2012 - Targeting User's Pocket and Privacy

Similar to previous years, Android was by far the most commonly targeted smartphone operating system in 2012, constituting 95% of unique threats [42]. These threats were found mostly in Eastern Europe and Asia though the number of threats in the rest of Europe and the United States had increased from the previous year. This year witnessed 350,000 malicious and high-risk Android apps comparing with less than 1300 malware in 2011 [43]. Also, they had an increase in the sophistication level. Furthermore, 103 Android mobile malware families were discovered in 2012 [44].

While stealing users' sensitive information - known as privacy leakage - was still among the main goals of malware authors, the primary focus changed significantly in this year. In particular, developing premium service abusers and adware outnumbered other types of Android malware. In general, 48.58% and 38.30% of detected Android malware in 2012 were service abusers and adware [43] [45].

Commonly disguised as popular legitimate apps, premium service abusers were designed to convince users to install them at the first step. Later, they could charge smartphone user by sending text messages to premium-rate numbers. Moreover, special variants could download other malicious apps, and, also, steal information from infected devices. For instance, variants of FakeInst and SMSBoxer families spoofed several popular Android games and social networking apps, including "Bad Piggies", "Angry Birds Space" and Instagram [46] [47]. Also, Gappusin variants downloaded other malicious apps and stole information from compromised devices [48].

Aggressive adware - ranked as the second popular types of malware in 2012 - could also disclose user's or device's sensitive data to third-parties by continuously showing advertisements to the smartphone user. These malware posed a high privacy risk to users and could gain big profits by selling users' information to untrusted parties. For instance, variants of Plankton family were able to collect personal information such as email addresses and phone numbers, and, could later, forward them to several third parties [49].

2.4.4 2013 - The Evolution of Advanced Trojans

While premium service abusers and adware remained the most common Android threats in 2013, four important changes happened all of which affected the Android malware evolution. First of all, the number of smartphone users grew in 2013 most of which were less aware of security risks of Android apps [44]. Second, Google's official app store, known as Google Play Store (introduced in 2012), superseded Apple's App Store and could become the largest smartphone app market at that time [44] [50]. This increase in popularity was later abused by malware authors. Third, an important vulnerability, called "Master Key", was found in July 2013 that allowed installed apps to be turned into malicious without user's consent nor knowledge [51]. Last but not least, several toolkits started to be sold for trojanizing Android apps in underground markets [44] [50].

Android continued to be the most targeted smartphone operating system in this year [50]. F-Secure and Trend Mircro reported 358 new Android malware families and 1 million new Android threats in 2013 alone [50] [51]. On the other side, malware authors put their focus on popular applications in online stores, especially games and social networking apps, to maximize the number of victims [43] [52]. Thus, trojanized versions of popular apps made up most of the additions to 2013's list of Android malware families [51]. In particular, advanced Trojans, adware and rogue security apps were among special interests of malware developers. Furthermore, spreading malware through email and fake websites were used commonly.

Among Trojans detected in 2013, Stels, Obad, Dandro, Fakeguard, Fakedefender and Pincer were the most popular ones. Stels was developed mainly to target smartphone users in Russia. Initially detected in 2012, F-Secure identified over 1,300 unique Stels samples that could be divided into three major variants as of 2013. It was the first Android malware distributed via spam e-mails, and a bot that used Twitter to update its C&C server addresses [44] [50]. Also, it was one of the pioneers developed to target multiple platforms (Android and Windows). Stels could act as a banking trojan and intercept incoming text messages to steal mobile Transaction Authentication Numbers (mTANs), thus defeating the two-factor authentication method used to validate an online banking transactions [50]. In addition, it could leak sensitive device information, including IMEI and IMSI.

The second common Trojan, Obad, could exploit "Master Key" vulnerability to elevate its privileges at the first step [44] [50] [51]. Once exploited, malware developers were able to embed their malicious codes into the modified apps. An interesting point was that malware developed this way resembled legitimate applications in terms of their signatures [44].

Discovery of Obad Trojan revealed that mobile threat landscape had evolved a lot since 2012 and that attackers had found new methods to compromise devices. For instance, Dandro was a Trojan developed using a toolkit, called binder, which was available for developing Remote Access Trojans (RATs). This malware began to circulate using email attachments with which the remote attacker was able to perform a wide variety of operations, including sending and retrieving text messages, retrieving contacts and call logs, acquiring the device location and using its camera to name a few. Later, a variant of Fakeguard family was detected which was a repackaged version of a legitimate Korean app. Once installed on the device, the app attempted to download its malicious code through sending update notifications to users [44]. Fakedefender was another advanced Trojan and the first Android malware that came into existence by leveraging security apps on smartphones. This rogue security app tried to persuade users into paying a considerable amount of money to get rid of all non-existent (or fake) threats from their device [44]. Finally, variants of Pincer and FakeKRBank were found that stole text messages containing TANs similar to Stels and were developed mainly to target users of the Commonwealth Bank [50].

2.4.5 2014 - The Emergence of Ransomware

Cybersecurity firms observed interesting changes in Android malware during 2014. On the one hand, the number of variants per family dropped by 16%. In 2012, 38 new Android variants per family was detected. This number increased to 57 per family in 2013, and it decreased again to 48 in 2014 [53]. One the other hand, 17% and 36% of Android apps were found to be either malware in disguise or grayware. While not malicious by design, Android grayware could annoy users by doing unintentional activities such as tracking user's behavior [53].

This year was undoubtedly the first time where different types of Android ransomware and bitcoin-mining malware were discovered [53] [54]

[55] [56] [57]. At this period, cybercriminals looked at developing ransomware as a lucrative business, and bitcoins became the payment method of choice by most new ransomware because of its strong anonymity. In particular, some ransomware got the press, including Koler, Slocker, Simplelocker, CryptoLocker, CryptoDefense and CryptoWall.

Koler was almost the first discovered Android ransomware [54]. It was the mobile extension of "police-themed" Reveton ransonware. This app was assumed to offer access to adult contents. However, once installed on the device, it started to send notifications to scare the user into paying a fine for supposed illegal activity [55]. Koler did not encrypt any files despite its claims; it only disabled the Back button to keep the ransom demand prominent.

Slocker - the first Tor-encrypted ransomware - could disable the Back button as well. However, it had much more capabilities. For instance, variants of Slocker were able to encrypt images, document and video files as well as communicating with a remote server via Tor network or SMS messages [54]. Simplelocker was a similar ransomware detected in June 2014 with file encryption capabilities. It employed an FBI social engineering theme like the famous Porndroid ransomware [53] to deceive users. However, Porndroid could take the victim's picture to display it later alongside the ransom demand. Later on, other ransomware were also spotted which made use of a Tor component such as Torbot, Dendroid, CryptoLocker, CryptoDefense and CryptoWall [57].

Despite developing these advanced Android ransomware, malware developers began to explore other ways through which they could extort money. Soon after that, they started to target Network-Attached Storage (NAS) devices, where large quantities of files were stored. For instance, Synolocker was developed to target Synology NAS devices by using a previously unknown vulnerability in Synology's DiskStation manager software. Once found its way into the device, it could encrypt all files, holding them for ransom [53].

In addition to developing different types of ransomware, campaigns had special attention to bitcoin-mining malware as well. As an example, Kagecoin could start mining for bitcoins on Android devices as soon as it was installed [56]. This allowed cybercriminals to use infected mobile devices' computing resources to mine for bitcoins and other cryptocurrencies. Fur-

thermore, infection resulted in shorter battery life, which could ultimately lead to a shorter device lifespan.

2.4.6 2015 - Charging Users by Sending Text Messages

While ransomware remained frequent in 2015, sending text messages to premium-rate numbers was the most common behavior observed among the top 10 Android malware families [58]. In particular, SmsSend, Slocker, FakeInst, GinMaster, GingerBreak, SMSpay, DroidRooter, Dialer, SMSKey and Coudw made up 25% of the total amount of Android malware detected in 2015 [58].

SmsSend, FakeInst, SMSpay and SMSKey were popular Trojans that charged users' phone bills by sending text messages to premium-rate phone numbers or a subscription-based paid service. Also, variants of these families used randomization techniques to evade detection by antivirus products [59]. Throughout this year, France was intensively attacked by SmsSend Trojan family [58].

In addition to SMS-sending Trojans, Android security flaws were exploited to lunch new attacks against users in 2015 [60]. For instance, Android's MediaServer component took a lot of hits as its vulnerabilities could be exploited to perform attacks using arbitrary code execution. These attacks could force a device's system to go on endless reboot in order to drain its battery. They could even be used to render Android devices silent and unable to make calls due to unresponsive screens [60]. Other vulnerabilities exploited in this year were related to Android debugger and installer components. A vulnerability in the former component was utilized to expose a device's memory content, while a hijacking vulnerability in the second component gave hackers the ability to replace legitimate apps with malicious versions in order to steal user's information.

2.4.7 2016 - Looking for Super-User Privileges

Android vulnerabilities increased by 206% [61] in 2016; however, attacks slowed for the first time since attackers faced security improvements in the Android's architecture [62]. Specifically, 4 new Android malware families and 3.6K new variants were found in this year. On the other hand, new ways were devised to bypass Android protection mechanisms [63]. For instance, variants of the Tiny SMS Trojan were detected that were able to

use their own window to overlay a system message warning users about sending a text message to a premium-rate number.

The year's most prevalent goal among Android malware was focused on gaining as much privileges as possible [63] on the victims' devices. This was achieved using one or more than one application.

In order to obtain super-user privileges using one single application, Trojans with almost unlimited capabilities were developed. These Trojans could install other advertising apps or malware stealthily. Representatives of this type of malware had been repeatedly found in the official Google Play app store in 2016 (e.g., Ztorg.ad was a Trojan that pretended to be a guide for Pokémon Go application).

App collusion was a mean to obtain the maximum level of privileges using more than one application. Here, instances of app collusion were discovered in a group of applications that used a particular Android SDK [64] [65] [66] [67] with a wide range of permissions. Co-working together, any of applications participated in the app collusion, could bypass the Android OS limitations and respond to commands from a remote server via the app with highest privileges. This could result in the maximum elevation of privileges and the most capable bot functionality based on all apps that took part in the negotiation.

In addition to the attempts made in gaining super-user privileges, a considerable growth was observed in ransomware [63]. This rise was caused by the active distribution of two families of Android ransomware, including Fusob and Congur. Fusob displayed the ransom demand on top of other windows, thus making it impossible to use the device, while Congur set or reset the device passcode and gave attackers administrator rights to the device.

Malware products and services such as malware kits offered on Dark Web marketplaces boosted the rise of Android malicious apps in 2016 as well. For example, DroidJack was offered by different vendors on four major marketplaces. This popular Remote Access Trojan (RAT) was sold openly on the Clearnet and Dark Web. The Android bot rent service (BaaS, or Bot as a Service) was also available for purchase. The bot, which was available both in Russian and English, could be used to gather financial information from Android smartphones [63].

2.4.8 2017 - The Rise of Android Screen Lockers

New mobile malware variants grew by 54% in number in this year comparing with the previous one. However, the act of rooting Android devices decreased as newer versions of Android OS provided increased functionality [11]. Also, the proportion of Android devices not being encrypted fell down for the first time. An analysis of Android smartphones revealed that only 20% of devices were on the latest major release [11].

Once again, the easy availability of exploit kits and dark web sources boosted the rapid creation of new Android malware [68] [69] [70] [71]. In particular, new ransomware rose by 36% in the second quartile [68], largely from widespread screen-lockers [68]. Also, this year witnessed serious vulnerabilities exploited to provide attackers with root privileges (e.g., Dirty COW vulnerability exploited by ZNIU malware) [72].

2.4.9 2018 - Android Malware Predictions

Threats to mobile devices will increase significantly in 2018 by the widespread usage of mobile phones for surfing the internet and the growing rate of mobile transactions [73]. However, Android will still be the most popular target platform for attackers because of its high market share. Specifically, more high-end APT malware will be discovered due to an increase in the number of attacks, and, also, improvements in the security technologies [74].

Security experts believe that advanced types of ransomware such as RaaS will emerge targeting users' sensitive information. in addition, Android malware will explode on Google Play according to their inspections [75].

Hijacking cryptocurrencies will likely rise in number as well due to their significant worth at the moment. Therefore, malware writers would try to secretly mine cryptocurrencies without users' knowledge [73].

2.5 Android Malware Research Datasets

This section presents the most prevalent Android malware datasets used in academic research works in the recent years. Although several datasets

might be available containing different Android malware families and variants, Malgenome [39], Drebin [76], Contagio Mobile Mini-dump [77], PRAGuard [78] and AMD [79] are the most popular ones. Also, AndroZoo [80] is one of the biggest available repositories of Android apps released recently for Android application and security analysis.

2.5.1 Malgenome

Malgenome consists of 1,260 malware samples from 49 different families discovered from 2010 to 2011. These specimens are obtained by carefully examining the security announcements, reports and blog contents. From this collection, 5 families are discovered from official Android market, while 35 families are found from alternative Android markets. Malware families in Malgenome are further investigated by the way they are installed and activated.

Regarding installation techniques adopted, almost 41% of families (20 out of 49) contain samples which are repackaged versions of legitimate applications. Thus, malware writers embed their malicious payloads after disassembling popular benign applications from official stores. 6% of families (3 out of 49) contain drive-by download malware. This group of malicious apps try to deceive users to download and install some risky applications mostly by in-app advertisements. 2% of Malgenome families are installed through update attacks where malicious payloads are downloaded at runtime to keep the whole payload enclosed until it is really needed. 41% of families (20 out of 49) contain standalone malware, and 18% of them use more than one installation technique.

Malgenome families make use of 9 different activation techniques. The most popular time where malware writers activate their malicious apps is when the booting process is completed successfully. This is a perfect time for malware to start its background services. Also, many malware specimens can be found which are activated upon receiving a call or a text message. On the other hand, few activation techniques (e.g., activating malware upon user's clicks on the app icon or the smartphone's home screen) are less frequently adopted by malware families in Malgenome dataset.

A big amount of Malgenome malicious apps make use of three techniques to compromise users' devices. First of all, piggybacking legitimate

applications on official Android markets is commonly observed in malware families. Second, communicating with remote C&C servers is used in the majority of apps to execute the intended commands. Last but not least, anti-analysis techniques in general, and obfuscation in particular is applied to malware samples in order to hide their malicious payloads and to obscure their semantics.

2.5.2 Drebin

Drebin malware dataset is a collection of 5,560 Android malware samples from 179 various families collected from 2010 to 2012. This repository contains all the specimens which were previously released in Malgenome dataset. The authors of Drebin make use of two sets of features to identify Android malware from benign applications. The first set of features are extracted from the app's manifest file, while the latter set of features are extracted from the disassembled code of application. Hardware components, app's components, requested permissions and filtered intents are the features which are extracted from the manifest file, whereas restricted API calls, suspicious API calls, used permissions and network addresses are obtained from the disassembled code. Afterwards, an SVM classifier is trained to tell apart malware from benign applications.

Analyzing the contribution of these features in the detection of Drebin malware families reveals that few of them such as requested permissions and suspicious API calls are not enough alone to distinguish malware from benign applications as they are prevalent in both set of apps. On the contrary, some of them, including used permissions and restricted API calls are only present in a small subset of families.

2.5.3 Contagio Mobile Mini-dump

Contagio Mobile Mini-dump dataset is part of a blog[1] where you can upload your malicious apps, and, also, download the ones already available here. It has started its activity from Jun 2011 and has been continuously gathering and reporting new malicious apps since then.

As of October 2018, it contains 363 zip files which are presumably its Android malware families which all add up to more than 5 GB in size.

[1] http://contagiodump.blogspot.com/

From this number, 13 zip files or families cannot be downloaded successfully due to crashing. However, it contains a wide variety of Android malware from Trojans to spyware and ransomware, and, thus, have been used in many research works till today.

2.5.4 PRAGuard

This dataset [81] is a collection of Android apps obtained from obfuscating Malgenome and Contagio Mobile Mini-dump applications with 7 different techniques. PRAGuard is composed of 10479 obfuscated applications from more than 50 different malware families.

Obfuscation methods applied to the malicious apps are considered as any modifications of either the app's .dex file or its .xml files such as it AndroidManifest.xml file. Thus, these approaches fall into two main categories, called trivial and non-trivial obfuscation techniques. From trivial category, identifier renaming has been applied to change the key identifiers of Android malware, including class names, method names and field names, package names and the names of source files. From non-trivial category, reflection, string encryption and class encryption have been utilized.

2.5.5 AMD

Android Malware Dataset (or AMD) contains 24,650 Android malware samples classified in 71 different families and 135 sub-families (known as varieties). These malicious apps are discovered in the period of 2010 to 2016. Malware family names are obtained by extracting a dominant keyword from the outputs of anti-virus engines in VirusTotal [82], whereas malware behavior groups (or varieties) are identified by applying a clustering technique based on the apps' malicious payload mining.

AMD authors have categorized malware families in a variety of ways similar to Malgenome dataset, including the way they are composed, installed and activated, the type of information they steal, their persistence level, the way they communicate with the remote server, the privilege escalation techniques they use and the anti-analysis methods they adopt to hide their logics.

AMD considers three different methods by which an Android malware can be composed. Based on this, malicious apps are either written from the

scratch (standalone malware) or they are repackaged within legitimate applications. The third group of apps are library apps that are composed by embedding malicious components in the library code of legitimate apps. Results show that 85 malware varieties ($\approx 63\%$) contain standalone malware, 40 varieties ($\approx 30\%$) consist of repackaged malware and only 9 varieties ($\approx 7\%$) contain library apps.

Android malware can be installed in two different ways according to the AMD authors. In the first way, known as dropping, the malware specimen tries to download and install some risky applications on the victim's device, whereas in the second way, called drive-by download, malware is delivered to the target device either without user's knowledge about the download procedure or about its consequences. 76 different varieties are installed by dropping comparing with only 15 varieties which are installed through drive-by download.

In addition to event based activation methods considered by Malgenome and Drebin authors, two extra activation techniques are observed in the Android malware of AMD dataset. Based on this, an Android malware is either activated by a host application or in a previously scheduled time. The latter technique is highly prevalent in Android ransomware.

The types of information which are stolen by AMD Android malware are related to the device or to the users themselves. International Mobile Subscriber Identity (IMSI), network operator, International Mobile Equipment Identity (IMEI) and device ID are some unique device information, while browsing history, contact list, bookmarks and calendar events are some personal information assigned to smartphone users. In AMD, 92 different malware varieties steal device information, whereas only 62 varieties steal personal information.

AMD malware samples make use of two general techniques to be as persistent as possible on the victim's device. The first approach tries to make the malware's presence as stealthy as possible on the target device. This is achieved in different ways, e.g., by blocking the appearance of special items such as calls and text messages, by cleaning some clues such as calls log and SMS history, by hiding the app icon, and, finally, by hooking system APIs to mask the malware's existence. The second approach prevents the malware from being destroyed by the system, user and anti-virus products. This is done by hiding the malware to be appeared in the de-

vice's administrator list, killing anti-virus processes or locking the device to name a few.

Nowadays, malware samples try to escalate their privileges to have the highest possible impact on the target device. Privilege escalation is achieved either by acquiring admin privileges or by using root exploits. Some privileged operations include locking the device, wiping device data and changing its lock-screen pin code all of which make the malware harder to remove and more persistent. For instance, Obad and Fobus, two famous Android backdoors try to get admin privileges, while Lotoor and Triada, two sophisticated malware families, try to obtain root privileges.

Communicating with a remote command and control server increases the flexibility and functionality of the malware. Android malware that receive their commands from C&C servers are composed of two main modules, including a message builder and a command handler. Also, they use a variety of formats to save and send their information to remote servers. For example, SMSZombie and FakeAngry embed their information in text messages and URL links, whereas SpyBubble and RuMMS encode their information in XML and JSON file formats respectively.

AMD malware families make use of various anti-analysis techniques to hinder their static or dynamic analysis. Identifier renaming, string encryption and dynamic loading of Android APK dex files are the most popular methods taken by malware samples in this dataset though other less frequent approaches such as evading dynamic analysis and using native payloads are also adopted.

2.5.6 AndroZoo

AndroZoo is the biggest available dataset of Android apps released to date [80]. Published formally in 2016 by the researchers in Université du Luxembourg, it contains more than three million unique samples, adding up to more than 20 TB in size. This collection has been created by crawling various Android app markets from 2011 and is continuously growing in size since then. As of October 2018, AndroZoo has more than 7 million Android apps. Each of the samples in this dataset are regularly analyzed by numerous anti-virus products to know which applications are detected as malware.

The majority of samples (96.88%) are gathered from 3 main Android app markets, including Google Play, Anzhi and AppChina. A recent study shows 1%, 33% and 17% of these three markets are malicious apps based on the results of at least 10 different anti-virus products [83]. In total, AndroZoo contains apps from 13 different app markets including some open-source repositories such as F-Droid [84].

2.6 Android Malware Analysis

Accurate analysis of Android malware is an important yet challenging task. Easy reverse engineering of Android apps is a double sword. On the one side, malware writers can make use of a wide variety of tools to embed their malicious codes into legitimate and popular apps. On the other hand, the same tools can be used to analyze Android malware rapidly and without additional costs. Although most of these tools have flaws and cannot precisely retrieve apps' information, they can capture the majority of apps' features that can be useful in their analysis.

In addition to visual inspection of malware using reverse engineering tools, two other approaches have been frequently used for Android malware inspection in recent years. In the first approach, information flows are used to model apps' behavior, while in the second method, pattern mining and data mining are used to characterize and study malware samples. Therefore, in what follows, each of these three ways of Android malware analysis are discussed briefly.

2.6.1 Reverse Engineering Tools

Android reverse engineering needs the main app's source code (stored in the .dex file) to be either disassembled or decompiled. Based on this, several tools can be used that are presented in what follows.

2.6.1.1 Disassemblers

Apktool is probably the most popular tool proposed for Android apps decoding and disassembly [29]. Applications utilize code and resources, known as framework resources, which are used by this tool to decode apk files.

Apktool has a number of advantages that have boosted its popularity in addition to the great documentation it is accompanied with. First of all, it is a multi-platform tool that can be easily installed on Linux, Windows and Mac operating systems. Second, it can be used to decode Android apps' resources to original form and rebuild them after modifications. Last but not least, Apktool can transform (disassemble) the binary Dalvik bytecode (.dex file) into the Smali code.

Baksmali is another well-known disassembler which uses dexlib library to read .dex files and the StringTemplate library to generate the disassembly. Baksmali's syntax is based on Jasmin/dedexer and supports the full functionality of the dex format. This tool is also supported with a well and up-to-date documentation.

2.6.1.2 Decompilers

Android malware analysts may decide to convert .dex files to Java using Java decompilers for further inspection. Decompiling process can be done either directly or indirectly. In the former way, called Java decompilation, .dex file (Dalvik bytecode) is converted to .jar file (Java bytecode) initially. Then, any available Java decompiler tools can be used to convert Java bytecode to Java code. In the latter way, known as Dalvik decompilation, .dex file is converted to Java source code without any intermediate processing.

Dex2jar is a multi-platform tool that enables analysts to convert Dalvik bytecode (DEX) to Java bytecode (JAR) [27]. Also, it allows them to use any existing Java decompilers with the resulting JAR file, including Jd-gui [85], JAD [86], Dare [87], Mocha [88] and Procyon [89] to access the Java source code of each application.

DEX to JAR transformation loses important metadata that the decompiler could use. Thus, pure Dalvik decompilers skip this step, and, as a result, they produce better output. However, there are not many choices available for Android as it is for Java. DAD [90] and JEB [91] are two open source and commercial Dalvik decompilers respectively which convert .dex files directly to Java source code.

2.6.2 Information Flow Analysis

Any information flow has two critical points defining its direction, known as source and sink. Sources are points within the program where sensi-

tive information are collected and commonly stored in memory, whereas sinks are points in which such data are leaked out of the program. Information gathering and leakage are both performed using a wide range of available Application Programming Interface (API) methods in the Android platform. For instance, an information flow from getDeviceId() API call to sendTextMessage() API call found in a malware specimen would imply that it leaks our device's unique identifier information to a remote server by sending text messages. Thus, these points can significantly help in understanding the underlying semantics of malware specimens.

Information flow analysis is a good and precise way of analyzing Android malware as it reveals how and why apps use specific pieces of information. It provides meaningful traces that represent how sensitive device or user information are propagated amongst the variables (and components) of a program.

Information flows have different variants. From one point of view, they are divided to explicit and implicit flows. The former category analyzes data-flow dependencies without considering the program's control-flow, while the latter considers the control-flow of the program. From another point of view, flows are categorized as inter-app and intra-app based on the type of communication. Inter-app flows are established between components of different applications, whereas intra-app flows establish between components of the same application.

Labeling sensitive data and tracking the way through which they propagate in an Android program, known as taint analysis, in known to be an effective way of extracting information flows. Taint analysis can be performed in either a static or dynamic way. Static taint analysis has a relatively low runtime overhead though it is imprecise. Also, it does not scale well with the number of applications. Moreover, it can be bypassed easily using advanced obfuscation techniques. On the contrary, dynamic taint analysis is more accurate as it models runtime behavior though it may miss parts of the code that are not exercised explicitly. Also, it usually imposes a high overhead to the system. Furthermore, apps can fingerprint dynamic monitoring systems to evade detection, and, thus, hinder dynamic taint analysis. FlowDroid [92], DroidSafe [93], FlowMine [94], CHEX [95], LeakMiner [96] and AndroidLeaks [97] are the most popular static taint analysis tools, while TaintDroid [98] and DroidScope [23] are the most

precise dynamic taint analysis tools proposed for information flow extraction and Android malware analysis.

2.7 Anti-Analysis in Android Malware

The openness of Android operating, in conjunction with its high popularity, has made it an attractive target for malware campaigns. As a reaction to this and to alleviate security and privacy concerns, several app analysis tools and services have been developed. Google has also joined this attempt by introducing Google Bouncer [99], a service that automatically scans Android apps to find potentially hidden malicious activities. Apps scanning is performed either in a static or dynamic way. Although both of these approaches provide useful information about apps behavior, they can be evaded by advanced anti-analysis techniques, also known as app hardening methods [100]. Anti-analysis products and services are commonly provided as Software Development Kits (SDKs) with binary libraries.

Furthermore, anti-analysis techniques are used frequently by both malware writers and legitimate app developers in the Android platform. In the first context, they are used to hide the apps' semantics from analysts by increasing the cost of reverse engineering, while in the second context, they are used to protect apps from illegal cloning or copying [101] [102].

From one point of view, anti-analysis techniques used by Android apps are divided into two main categories; those which are used to evade static analysis, and the ones which are applied to hinder dynamic analysis of apps [103]. Identifier renaming, string encryption, packing, code obfuscation and information hiding are some strategies which are used from the first category. However, these methods may not necessarily be successful in breaking the static analysis [104]. On the contrary, the tools in the second category focus on detecting debuggers, virtual machines, sandboxes, emulators and other runtime monitoring tools. For instance, certain types of malware may be found which stay inactive if they observe certain identifiers they have already implanted in execution environment such as browsing history or cookies in order to evade sandboxes [105].

From another perspective, anti-analysis methods are classified into three major groups [106]. The first group of methods try to evade detection using static information initialized to fixed values in the emulated environment (e.g., Device ID, build version of Android SDK and the layout

of routing table). The second group try to do so by observing unrealistic behavior of different sensors (e.g., accelerometer, gyroscope, GPS), and, the last group, do this based on incomplete emulation of the actual hardware (e.g., by identifying the QEMU scheduling). Different behaviors of QEMU scheduling points on an Android emulator and a real device has already been studied and confirmed [106].

2.8 Data Mining Tools for Malware Analysis

The process of discovering hidden patterns from a big pile of data, or, in other words, getting an insight about the data stored in databases is commonly known as data mining [107]. The data can be stored electronically, and the search for patterns is commonly automated by computer. Data mining has been used in a variety of domains, including many areas in cybersecurity [107], and, specifically, in malware detection [108]. There are basically two general approaches for mining meaningful data from big databases.

Traditional data mining algorithms need to have the whole set of past observations (referred to as the training set) to discover interesting patterns and will be used later by machine learning algorithms to predict future observations. Thus, to explore new patterns from a new set of observations, they need to be re-run. However, with the emergence of new devices and technologies, and the amount and frequency of data generated by them such as smartphones and the Internet-of-Things (IoT), traditional data mining algorithms cannot be applied efficiently as they need to be repeated in short intervals that is not feasible at a low cost.

Continuous and fast streams of data introduce big challenges to traditional data mining algorithms in particular, and machine learning methods working based on them in general. Some of these challenges include but not limited to concept drift [109], feature drift [110], temporal dependencies [111], and restricted resources requirements, both in time and memory. In addition, typical issues known in traditional data mining and machine learning algorithms, including non-representativeness of training dataset, missed feature values, underfitting, overfitting, and irrelevant features may be found here. Thus, several attempts have been made in recent years to introduce new methods for handling data streams.

Data Stream Mining (DSM) is a variation of traditional mining techniques which tries to explore patterns from continuously and rapidly evolving data. The two approaches are similar in terms of predicting a label for new upcoming instances represented by a number of features known as feature vector. However, DSM methods build their models from an incrementally growing pool of training instances in contrast with a large static training dataset which is commonly used by traditional data mining algorithms [112]. Machine learning methods which are based on traditional data mining are known as batch learning algorithms, and the ones which make use of data stream mining are referred to as online learning algorithms. Due to the extensive application areas of DSM, several tools have been developed, including Massive Online Analysis (MOA) [113], Scalable Advanced Massive Online Analysis (SAMOA) [114], Advanced Data Mining and Machine Learning System (ADAMS) [115], JUBATUS [116], Vowpal Wabbit [117], StreamDM [118].

Online learning algorithms update their models over time (incremental learning) based on new coming instances compared to batch learning methods that keep their built model static once it is extracted. Therefore, online learning can save a significant amount of computational resources, and, also, the time which is taken for extracting the models. Furthermore, online learning algorithms do not require to decide on the number of instances to be used for training which is critical in the performance of batch learning algorithms. In return, they split the stream into disjoint chunks of data known as landmark windows. A landmark can be defined as the number of observed instances up to the moment. Thus, once a new landmark is reached all past instances are discarded. Another strategy is to discard one instance at a time which is done by sliding windows.

As a specific area of data mining, pattern mining has been commonly used to facilitate Android app analysis in recent years. One of the main areas where pattern mining algorithms have been applied is for smartphone usage prediction [119] [120]. Most of the works in this area have focused on mining behavioral patterns (or profiles) from Android applications using different features. For instance, a mining algorithm has been proposed in [121] to extract temporal API usage patterns from client programs in order to help developers having a precise and complete understanding of the current libraries. A similar work [122] extracts time-constrained sequential patterns using mining algorithms to identify application usage patterns

on smartphones. ApMiner [123] relies on association rule mining of android apps in the market to identify co-occurrences of permissions and API methods. Based on this, it recommends specific permissions which need to be added when developers use special API methods in their programs.

Another area in which pattern mining algorithms have been adopted is malware characterization. For instance, in a recent work [124], information on apps descriptions, together with sensitive data flow signatures, have been used to characterize 3,691 malicious and 1,612 benign applications.

Commonly, data mining involves six different tasks from which classification, clustering and association rule mining are the most important ones. Classification is the process of generalizing a learned model or structure to new data, while clustering is the process of exploring groups of data with similar features without using a previously learned model [125]. Association rule mining however, discovers possible relationships or dependencies between different variables [126].

2.8.1 Classification

In machine learning and data mining, classification is the process of identifying the right category (from a set of categories) of a new instance (or observation) based on a previously learned model built on the basis of a training set of instances whose category is clearly known. When such a training set with correct labels is provided, the task is considered as a supervised learning process in machine learning [127].

Classification algorithms for malware detection and analysis are capable of classifying new specimens as either malicious or benign. Thus, they are composed of two main steps, including model construction and model usage. In the first step, several features such as API calls and strings are extracted to create a unique feature vector for each instance in the training set. Then, feature vectors and real labels for different instances are passed to a classification algorithm from which it can create a predictive model. Once this model is built, the classification algorithm can automatically classify a new sample as one of the two categories. To do so, it first extracts the same set of features from the new instance and creates a feature vector.

Several features have been considered in the literature to train and test classifiers for Android malware detection. These features are obtained either through static or dynamic analysis. Permissions [128], sensitive API

calls [129], call graphs [130] and control flow graphs [131] are some features that have been used for malware detection based on static analysis, whereas user interactions with apps is considered via dynamic analysis in another work to achieve the same goal [132]. Hybrid approaches have also been proposed to extract more precise features for Android malware detection [133] [134] [135].

Feature selection is another important process in malware classification. The process of selecting useful or informative features from a big amount of features is known as feature selection. The main reason behind applying this process after feature extraction is the redundancy of features. In many cases, features do have overlaps. Thus, removing one feature does not affect others significantly. Also, some features may found to be totally irrelevant. Therefore, removing them will not have a significant negative impact on the classifier's performance.

Filter, Wrapper and Embedded are three main methods which are commonly used for feature selection once features are extracted. In the Filter method, statistical approaches are used to choose the best subset of features. Thus, features are scored using some specific criteria at the first step. Then, they are ranked based on the acquired scores. Finally, a subset of features which have scores higher than a threshold are used as candidate features. In the Wrapper method, the actual classification algorithm is used to select the best possible subset of features. Here, different subsets of features are chosen consecutively, and the performance of classification on a sample training set is used to measure the quality of each subset and to finally choose the best possible one. Although often leads to the optimal feature subset, this feature selection method is computationally expensive. Finally, in the Embedded method, best features are selected while the model is being constructed by the classification algorithm. In terms of computational complexity, this method is in between Filter and Wrapper method and is considered as a limited greedy search algorithm.

2.8.2 Clustering

Clustering is the process of grouping instances into various categories based on some measure of similarity. Since categories of instances is not known previously, this task is considered as an unsupervised learning process in machine learning. Thus, one advantage of clustering over classifi-

cation is that it does not require a big amount of labeled instances as training samples. Also, it can help in exploring behavioral features. However, clustering instances is not always straightforward.

In the malware detection area, a cluster contains one group of samples with common behavior or characteristics, while being different from the samples in other clusters. There are basically two well-known approaches for clustering, including Partitioning and Hierarchical clustering. When instances constitute regular shapes in the feature space, partitioning algorithms such as K-means can be used for clustering. However, when instances form irregular shapes in the feature space, hierarchical methods are used. in the case of malware analysis, choosing an appropriate clustering algorithm depends significantly to the extracted features and their distributions.

Partitioning algorithms aim to group instances into clusters so that they have the shortest distances to the centers of clusters to which they have been assigned and the farthest distance to the instances in other clusters. Hierarchical clustering however, seeks to build a hierarchy of clusters, and, thus, falls into two main categories, known as Agglomerative and Divisive, based on the way the hierarchy is created. In the former method, which is a bottom up strategy, each instance is first considered in its own cluster, and pairs of instances are merged into a bigger cluster if it is determined that they have some similarities based on a specific criteria (called Linkage criteria). In the latter method, which is a top down strategy, all instances are initially considered in one single cluster. Then, this cluster is splitted into different clusters based one similarities of instances as the algorithm moves down the hierarchy.

2.8.3 Frequent Pattern Mining

Frequent pattern mining is a subset of a bigger process, known as association rule mining, which is an important task in data mining. Association rule mining looks for associations, correlations and causal relationships among different items in transaction databases. Also, finding frequent patterns that appear together is another important application of association rule mining.

One of the most important applications of association rule mining is to understand customers' buying habits by finding the correlations among

different items customers put in their shopping basket. However, it is also applicable to some areas of cybersecurity such as fraud detection, web log analysis and malware analysis to name a few.

Given a database with a number of transactions, frequent pattern mining determines all patterns that are present in at least a percentage of transactions. This threshold is known as minimum support value and can be set either as a constant number or a fraction of the total number of transactions in the database. A second important parameter is confidence which is an indication of how often a rule is found to be true, or, in other words, how much is the probability of observing two different items together in the total number of transactions.

There are two popular approaches for frequent pattern mining, including Apriori [136] and Frequent Pattern growth (FP-growth) [137]. The Apriori algorithm starts finding frequent itemsets from individual items in the database. It then extends these individuals to larger sets (called candidate generation) when they appear sufficiently often in the database. The main idea behind this algorithm is that when any pattern is infrequent, its superset should not be generated and considered as frequent. However, it requires multiple scanning of the database. On the contrary, FP-growth algorithm does not need candidate generation and avoids costly database scans. It uses a compact Frequent Pattern tree (FP-tree) data structure to extract frequent item sets.

2.9 Adversarial Machine Learning in Android Malware Analysis

Adversarial machine learning is an important field of study which lies in the intersection of computer security and machine learning [138]. The primary goal of this field is to increase the robustness of machine learning algorithms by facing them with different adversarial settings where attack is a major component.

Attacks on machine learning algorithms are divided into two main groups, known as poisoning attacks and evasion attacks which are performed in the training and testing phases of learning algorithms respectively [139]. Poisoning attacks try to influence, learn or corrupt the learn-

ing model, whereas evasion attacks force the model to produce selected adversary outputs.

Regardless of the type of attack, machine learning algorithms can be either fooled or evaded using a wide range of techniques at different cost. Recent studies also support that machine-learning-based algorithms can be significantly degraded in performance using carefully-crafted malware variants [104] [140] [141] [142] [143] [144].

The majority of these adversarial approaches challenge the robustness of Android malware detectors (or classifiers) by applying different feature manipulation techniques (e.g., removing, adding, modifying) upon which the models are built or trained. Most of these processes are focused to be performed in the testing phase in order to masquerade a malware as a benign application or to produce misclassifications. Applying these methods are straightforward as decompiling, manipulating and repackaging Android apps are all easy in Android platform [145] [146].

Easy repackaging of Android apps has thus forced security engineers to come up with effective solutions when designing their Android malware classifiers [143] [144] [147]. First of all, classifiers can be built based on some features that are difficult to be modified without affecting the apps' malicious functionality. This would leave less room for attackers to apply various kind of transformation techniques on features. Second, new methods can be used to make the feature ranking (or features' weights) process totally unpredictable, and, therefore, to make it more difficult for attackers to have any reasoning about the importance of features used by classifiers. Last but not least, retraining classifiers with different random set of features would increase their robustness against adversarial attacks.

3

TriFlow: Triaging Android Applications Using Speculative Information Flows

3.1 Introduction

The amount and complexity of malware in Android platforms has rapidly grown in the last years. By early 2016, both Symantec and McAfee report more than 300 malware families totaling over 12 million unique samples [59] [148]. Every malware family (and, sometimes, every sample within a family) may pose a different threat. The sheer number of apps available in current markets, along with the ratio at which new apps are submitted, makes impossible to manually analyze all of them. Automated analyses also have their limitations and some techniques might require a substantial amount of time per app [149]. This has motivated the need for a multi-staged analysis pipeline in which apps should be initially triaged to allocate resources intelligently and guarantee that the analysis effort is devoted to those samples that potentially have more security interest.

One of the salient features of Android's security model is its permission-based access control system. Apps may request access to security-and privacy-sensitive resources in their manifest file. These requests are presented to end users through permission dialogs at install time or, since Android version 6 (*Marshmallow*), at runtime for a reduced subset of permissions. Requesting access to protected resources is a clear indicator of risk and most triage systems for Android apps have relied quite heavily on requested permissions (see, e.g., [150] [151] [152] [153] [154]), since

they have proven effective to identify apps carrying malicious functionality. The majority of these approaches rely on metrics that combine the prevalence (or rarity) of each permission in benign and malicious apps with the criticality of the resources protected by the permission.

Using permissions alone to assess risk has important limitations [155]. Permission-based risk metrics might be highly inaccurate for two reasons. First, apps are often overprivileged and many permissions requested in the manifest might not be actually used during execution. Second, they assign a risk to a particular permission (e.g., INTERNET) just because it could be used as a vehicle for a malicious purpose, such as leaking out a piece of sensitive data, without considering if sensitive data is actually being sent or not. Determining risk using Information Flows (IFs), as done by the approach introduced in this chapter, overcomes this limitation and provides a more accurate assessment of the app's actual behavior. However, IF analysis presents a number of challenges. Identifying flows in an app involves a non-negligible amount of resources both in time and memory. For instance, according to our experiments, it can take more than 30 minutes per app to extract IFs from at least half of the samples in the Drebin dataset [156] using a relatively powerful computer (40 processors and 200 GB RAM). The situation may even be worse when analyzing apps with sufficiently large call graphs. In those scenarios, the IF extraction might not even be practical [157].

In this chapter, we describe TRIFLOW, an IF-based triage mechanism for Android apps that attempts to overcome the issues discussed above for permission-based systems, and, also, the limitations of existing IF analysis tools. Since extracting IFs from an app is an unreliable and computationally expensive process, TRIFLOW introduces the notion of *speculative information flows*. This means that TRIFLOW extracts some features from apps and then predicts the existence of a flow based on them. Prediction is done on the basis of a model that is previously trained using ground truth obtained with flow extraction tools. Each predicted flow is then scored by TRIFLOW in terms of its potential risk, which depends on the flow's observed prevalence in malware and benign applications. To do this, we rely on the cross-entropy between the empirical probability distributions of each flow in malware and benign apps. This provides a simple but sound quantification of the intuition that an information flow is risky if it is frequent in malware and rare in benign apps.

TRIFLOW has been implemented in Python and has been tested using a combined dataset of more than 17,000 apps. Our results suggest that it is possible to predict information flows efficiently with prediction errors remarkably small for the majority of information flows. The evaluation of the flow scoring measure reveals that 75% of information flows have no value at all for risk prediction, and only 1% of the remaining flows receive high weights. This suggests that malicious behavior (at least in the samples contained in our datasets) can be modeled using a relatively small subset of all possible information flows.

We evaluate TRIFLOW by simulating a triage process in which apps must be prioritized as they arrive. Experimental results demonstrate that TRIFLOW outperforms existing permission-based risk metrics in all considered scenarios. Additionally, TRIFLOW provides an explicative report that describes the flows that most contribute to the overall risk assessment.

In summary, contributions of this chapter can be listed as below:

- We introduce the idea of predicting the existence of a particular information flow using static features extracted from an app's code. We believe this idea might have potential beyond the scope of this work, and, more generally, could be extended to predict the presence of other program artifacts whose precise identification requires computationally expensive static or dynamic analysis procedures.
- We extend to information flows the notion of "rare equals risky" that has been largely explored and tested in the field of permission-based risk metrics. Based on this, we design an information-theoretic risk measure related to the cross entropy between the distribution of information flows in benign and malicious apps, thus quantifying how informative a flow is.
- Finally, we make our results and our implementation of TRIFLOW publicly available at

  https://github.com/OMirzaei/TriFlow

  to allow future works in this area to benefit from our research. TRIFLOW can be easily extended for new API methods and new information flows appearing in upcoming versions of Android, and its modular architecture facilitates its integration in existing risk assessment frameworks.

The rest of this chapter is organized as follows. Section 3.2 describes in details our approach for fast triage of apps based on speculative information flows. In Section 3.3 we present and discuss the results of our evaluation, including our prototype implementation and the datasets used (3.3.1). Additionally, we report: (i) the accuracy of the flow prediction (3.3.2) and the flow weighting (3.3.3) mechanisms; (ii) the triage results and the reports generated by TRIFLOW (3.3.4); and (iii) the efficiency of the tool (3.3.5). In Section 3.4 we discuss a number of issues and limitations of our approach. Finally, Section 3.5 discusses related work and Section 3.6 concludes the chapter.

3.2 Approach

This section describes our approach for fast triage of Android apps. We first provide an overview of our proposal in Section 3.2.1. We then describe its two key ideas: a probabilistic estimator for information flows (Section 3.2.2) and a weighting scheme based on the a priori risk contribution of each information flow (Section 3.2.3). This is later used to rank apps and prioritize analysis.

3.2.1 System Overview

A high-level view of TRIFLOW is provided in Fig. 3.1. The system is first trained using a dataset of benign and malicious apps. The goal of this phase is to obtain the two items that will be later used to score apps:

(i) A predictive model that outputs the probability

$$\theta_f(\phi_1, \ldots, \phi_n) = P[f \mid (\phi_1, \ldots, \phi_n)] \tag{3.1}$$

of each possible information flow f present in the app given a feature vector $(\phi_1, \ldots, \phi_n)$ obtained from the app's code.

(ii) A risk model consisting of a function $I(f)$ that measures how informative each information flow f is considering its relative frequency of occurrence in malware and benign apps.

The predictive model is estimated using both the feature vectors obtained from each app and the ground truth, i.e., the actual information flows

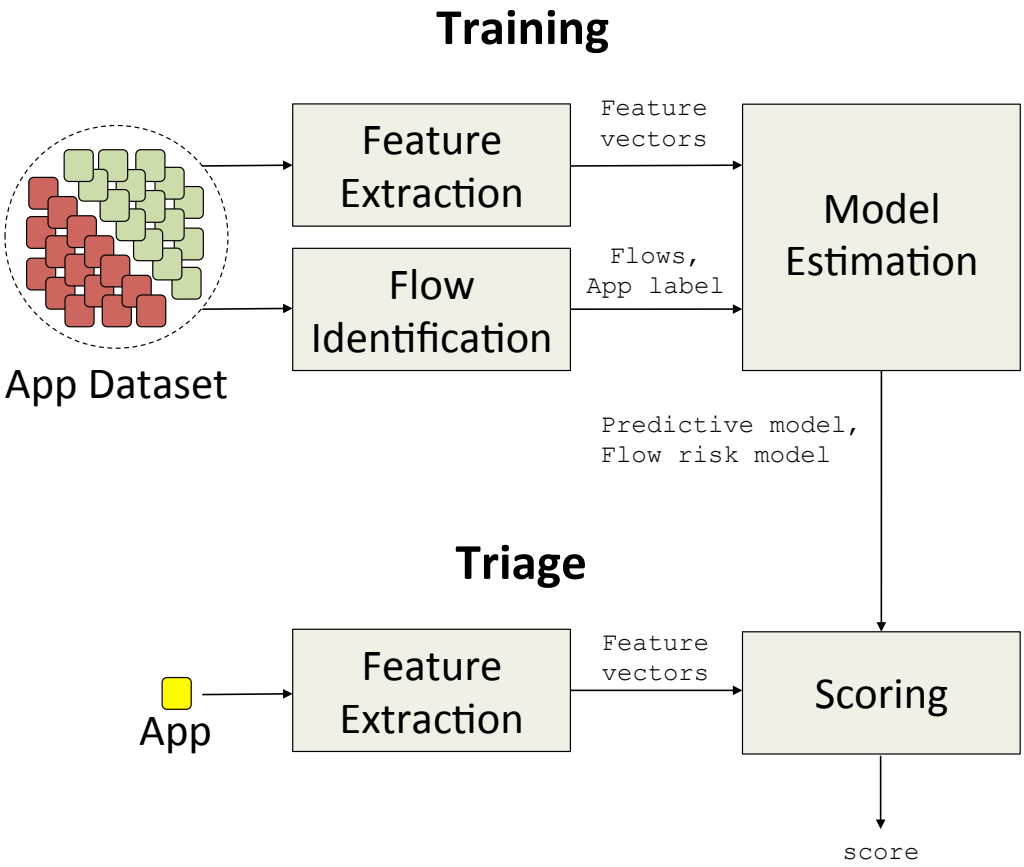


Figure 3.1: TRIFLOW Architecture.

present in the app; hence, the flow identification component in our architecture. Note that we also tag each flow with the app's label, i.e., whether it is benign or malicious.

Obtaining the score for an app (bottom part of Fig. 3.1) is done by simply multiplying each flow's likelihood by its weight and summing up for all flows:

$$\text{score}(a) = \sum_{f} \theta_f I(f). \tag{3.2}$$

Note that this only requires extracting the feature vector from the app and getting the θ_f and $I(f)$ values. As described in details later, in TRIFLOW both models (prediction and risk) are implemented as look-up tables, so the overall scoring process is extremely fast.

3.2.2 Predicting Information Flows

Let $f = (s, k)$ denote an information flow from source s to sink k. We aim at coming up with a predictor $P_f(a)$ that outputs whether f is present in an app a without actually performing an information flow analysis over the app. Our emphasis is on *efficient* predictors, so P_f has to base its decision on features that can be extracted very efficiently from the app. TRIFLOW uses the presence of a call to the source s and another to the sink k in the app code as features. Determining the set of sources and sinks called by an app is straightforward. It can be done very efficiently by simply decompiling the app's DEX file and matching the resulting code against a list of predefined sources and sinks.

We explored this idea using a probabilistic estimator as follows. Let $\mathcal{S}(a)$ and $\mathcal{K}(a)$ be the set of sources and sinks identified in the code of an app a. The set of all possible information flows in a is the product set $\hat{\mathcal{F}}(a) = \mathcal{S}(a) \times \mathcal{K}(a)$; that is, for each possible source $s \in \mathcal{S}(a)$ and sink $k \in \mathcal{K}(a)$, there is a potential flow $f = (s, k) \in \hat{\mathcal{F}}(a)$. We now assume that the occurrence of each flow $f = (s, k)$ in an app is given by a probability distribution $\Theta = (\theta_1, \theta_2, \ldots)$ where $\theta_f = P[f = (s, k) \mid s, k]$. The estimator can be obtained using a dataset $\mathcal{D}$ of apps (malicious or not) as

$$\theta_f = \frac{\sum_{a \in \mathcal{D}} \mathsf{ind}_f(\mathcal{F}(a))}{\sum_{a \in \mathcal{D}} \mathsf{ind}_f(\hat{\mathcal{F}}(a))}, \tag{3.3}$$

where $\mathsf{ind}_x(A) = 1$ if $x \in A$ or 0 otherwise, and $\mathcal{F}(a)$ is the set of actual information flows of the app a extracted using an information flow analysis tool. Note that the denominator in Eq. (3.3) is always greater than the numerator, since the presence of a flow in an app requires a call to both the source and the sink, and, therefore, such a flow will appear in the $\hat{\mathcal{F}}$ set.

Obtaining the θ_f estimator requires some computational effort since it involves obtaining the actual information flows for each app. However, once this task is done offline, the θ_f values can be stored in a look-up table and used after extracting the sources and sinks present in an app. Furthermore, the estimators can be incrementally refined when more apps become available, i.e., it does not require to go again through the set of potential and real flows for the already processed apps.

3.2.3 Informative Information Flows

The second component of our risk metric is a measure that quantifies how important a particular information flow is to distinguish malicious from benign apps. To do so, we adopt an empirical approach based on the relative frequencies of occurrence of information flows in both classes of apps. A similar idea has been leveraged by previous permission-based risk metrics such as [153] [154] [158], in which the risk of a permission depends on how rarely it is requested by benign apps. In TRIFLOW we implement this as follows. Let $P_M(f)$ and $P_B(f)$ be the probability of the information flow f occurring in malicious and benign apps, respectively. We seek to associate with f a weight $I(f)$ satisfying two properties:

1. $I(f)$ should be positively correlated to $P_M(f)$: the more frequent f is in malware, the higher $I(f)$. If f has never been observed in malware, i.e., if $P_M(f) = 0$, then $I(f) = 0$.
2. $I(f)$ should be negatively correlated to $P_B(f)$: the more frequent f is in benign apps, the lower $I(f)$. More specifically, if $P_B(f) = 1$ then $I(f)$ should be 0.

Both properties are satisfied by the following scoring rule:

$$I(f) = -P_M(f) \log_2 P_B(f). \tag{3.4}$$

Note that this score is essentially the probability $P_M(f)$ weighted by the $-\log_2 P_B(f)$ factor, which implements the negative correlation with $P_B(f)$. This factor can be interpreted in information theoretic terms as the self-information (or surprisal) of f when looked at from the perspective of benign apps (i.e., the P_B distribution). Incidentally, this provides a sound interpretation of $I(f)$ in terms of the cross entropy between the P_M and P_B distributions. Recall that the cross entropy between two probability distributions P_1 and P_2 is given by

$$H(P_1, P_2) = -\sum_x P_1(x) \log_2 P_2(x) \tag{3.5}$$

and measures the average number of bits needed to identify an event if a coding scheme based on P_2 is used rather than one based on the true distribution P_1. Thus, $I(f)$ can be seen as the contribution of flow f to the cross entropy between the probability distributions of flows in malware and benign apps.

3.3 Evaluation

This section reports and discusses our results. In Section 3.3.1 we first describe our implementation of TRIFLOW and the used datasets. The two core components of TRIFLOW are evaluated in Sections 3.3.2 (information flow prediction) and 3.3.3 (flow weighting). The effectiveness and efficiency of the overall triage mechanism are finally addressed in Sections 3.3.4 and 3.3.5, respectively.

3.3.1 Experimental Setting

TRIFLOW has been implemented in Python. Our implementation decompiles the DEX file using Baksmali and then scans the code searching sources and sinks in the smali representation. The list of sources and sinks is provided as an input and, in our current implementation, taken from the SuSi project [159].

To train and evaluate TRIFLOW, we have used two different datasets: (i) a set of real-world Android OS malware samples known as Drebin [156], and (ii) a set of goodware apps downloaded from Google Play at different points between 2013 and 2016. The malicious dataset (Drebin) was originally collected by Arp et al. [156] as an extension of the popular Android MalGenome project. The Drebin dataset contains 5,560 apps and about 171 malware families. Among other behaviors, the modus operandi of many of these specimens is largely related to fraudulent activities such as sending SMS messages to premium rate numbers. The benign dataset (GooglePlay) was retrieved from the Android official marketplace. It is comprised of 11,456 popular free samples downloaded from different categories, including popular apps such as *Facebook*, *Google Photos*, *Skype* or *MineCraft*. Table 5.1 summarizes both datasets.

The evaluation is based on two distinct and non-overlapping splits of the datasets, i.e., training and testing. The predictive model is extracted using the former, while the latter is used to perform triage over unseen apps. For training we retained 4,000 samples (71%) from Drebin and an additional set of 4,000 (35%) from GooglePlay. The training set thus contains the same amount of malware and goodware, i.e., a 1:1 malware-to-goodware ratio. Although the occurrence of malware in official markets

Table 3.1: Overview of the datasets used in this work. The upper part of the table shows the source of our dataset together with the number of samples from each source. The bottom part shows the training/testing splits used during cross-validation and the malware-to-goodware ratios.

Type	Dataset	Type	Samples
Malware (MW)	Drebin [156]	Malware	5,560
Goodware (GW)	Google Play	Goodware	11,456
		Total	17,016

Mode	Split	Ratio	Samples
Modeling (Training)	4,000 MW 4,000 GW	1:1	8,000
Triage (Testing)	1,560 MW 7,456 GW	1:5	9,016

is much lower than the presence of goodware, undersampling the training set is a common practice to equally weight both classes when building the model [160] [161]. For testing, we increased the malware-to-goodware ratio to 1:5, which is a common practice in other works in the area [129] [133] [157]. All these splits were done randomly and using a hold-out validation approach, i.e., the set of samples used for training differs from those selected for testing.

We then used FLOWDROID [92] to identify data flows in all apps in our dataset. We ran FLOWDROID[1] considering all Android API sources and sinks proposed in the SuSi project [159]. The extraction took place on a 2.6 GHz Intel Xeon Ubuntu server with 40 processors and 200 GB of RAM. We set a timeout of 30 minutes and between 40 GB and 100 GB of RAM per app in FLOWDROID. Even with this configuration, FLOWDROID could not finish the flow extraction process entirely for all the apps in our datasets. This lack of reliability has been reported before [157] and is indicative of the limitations (and computational cost) of techniques that rely on extracted information flows. For instance, analyzing a popular gaming app with more than 1 million installations in Google Play took about 90 GB of RAM and almost 2 hours of analysis time. Table 3.2 summarizes the main statistics of the dataset used to train TRIFLOW. In total, we identified 7,802 unique flows in the malware dataset and 28,163 unique flows in

[1] Version from mid 2016.

Table 3.2: Statistics of the training dataset. The size (in MB), number of sources (src), number of sinks (snk), memory consumed (in GB), and time (in seconds) are given on average per app. The amount of memory (in GB) required represents the maximum average.

#Apps	Size	#Src	#Snk	#Flow	Mem	Time
4,000 MW	0.9	150.5	100.6	63.5	14.3	55.0
4,000 GW	6.2	223.1	124.4	255.5	88.3	132.1
8,000 ALL	3.5	186.8	112.5	159.5	51.3	93.6

the goodware dataset. This difference can be attributed to the fact that apps in the benign apps set are, on average, much bigger in size and number of data flows than the apps in the malware dataset.

3.3.2 Flow Prediction Accuracy

Our first experiment evaluates the accuracy of the flow predictor introduced in Section 3.2.2. Our aim is to quantify the error made by the predictor and also to determine if such an error is somehow different for malware than for benign apps. Recall that θ_f provides the probability of flow f appearing in an app if the flow's source and sink are located in the app. We define the prediction error for f in an app a as

$$\text{error}(f) = \begin{cases} 1 - \theta_f & \text{if } f \in \mathcal{F}(a) \\ \theta_f & \text{otherwise,} \end{cases} \tag{3.6}$$

where $\mathcal{F}(a)$ is the set of actual information flows of a. The error defined quantifies how far from the true value (i.e., 1 if the flow appears, and 0 otherwise) the prediction is.

In order to obtain a robust estimation of the prediction error, we applied 5-fold cross-validation to the two modeling (training) datasets described in Section 3.3.1. We used non-stratified cross-validation, i.e., folds are randomly built. Thus, each dataset is split into 5 folds of approximately equal number of apps. In each of the 5 iterations we estimated θ_f using 4 out of the 5 folds and then obtained the error for all the apps in the remaining fold.

Table 3.3 provides the mean, standard deviation and median values for all the prediction errors obtained. In all cases, the results show that the

Table 3.3: Flow prediction error statistics after 5-fold cross-validation using only malware, only benign apps, and both.

Dataset	Mean	Std. Dev.	Median
Drebin	0.0861	0.1272	0.0278
GooglePlay	0.0361	0.0734	0.0094
All	0.0376	0.0784	0.0089

predictor works remarkably well. Interestingly, it seems to be slightly easier to predict flows for benign than for malicious apps. We elaborate on this later on in this section when analyzing prediction errors for individual flows. When combining both datasets, the average error is similar to the one observed for goodware. This could be attributed to the fact that malware specimens in our dataset are often repackaged [156] (i.e., the malicious app is built by piggybacking a benign app with a malicious payload), so many of the flows seen in malicious apps are not malicious as they do not originate in the piggybacked payload.

As for the provenance of the prediction error, Fig. 3.2 shows the error distribution for all flows in our datasets. We can observe that most flows are actually very easy to predict with low error. For the malware dataset, 4.31% of the flows (i.e., 337 out of 7802) are predicted perfectly (i.e., their prediction error is 0); around 83% of the flows can be predicted with an error lower than 0.1; and for around 90% of them the error is less than 0.25. The most frequent source API methods observed in these flows come from the TelephonyManager, Location, and Date packages. Similarly, the most relevant sink API methods observed come from the Camera.Parameters, and Log packages. For the goodware dataset, 1.04% of the flows (i.e., 293 out of 28163) are also predicted with no error and the figures are similar to the case of malicious apps (i.e., more than 90% of the flows can be predicted with an error lower than 0.25). Here, we observe that the most relevant source API methods come from Intent, Bundle, File, AudioManager, and View packages, while the most relevant sink API methods come from AudioManager, MediaRecorder, Log, Intent, and Bundle.

On the other hand, we observed a number of flows that are very hard to predict. In the case of malicious apps, flows from source methods used to retrieve data from intents (e.g., getIntExtra(java.lang.String,int)) to sinks related to media (such as setVideoEncodingBitRate(int)) are error prone.

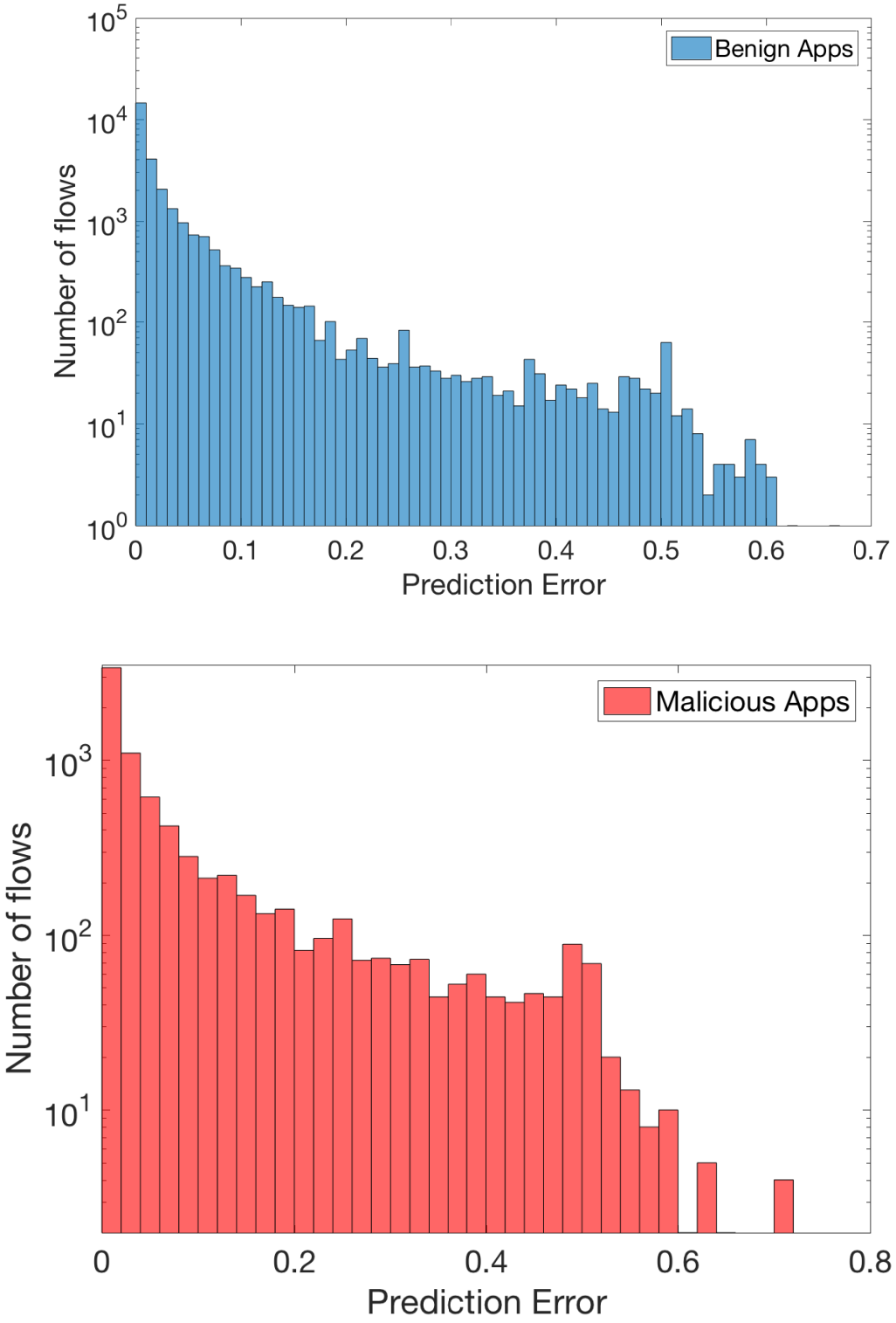


Figure 3.2: Distribution of the prediction errors for all information flows in the two datasets. Note that the in both plots the y-axis is in logarithmic scale.

For benign apps, we observe difficult-to-predict flows from sources that are used to retrieve PendingIntent before starting a new activity to sinks which are commonly used to set an intent when interacting with widgets (setPendingIntentTemplate(int,android.app.PendingIntent)). We did not examine further the reasons for such errors in certain flows and decided to leave this question for future work.

3.3.3 Flow Weights

We calculated the $I(f)$ values for all the 31,175 unique information flows obtained from the training datasets. Fig. 3.3 shows the cumulative probability distribution computed over the obtained values. Around 75% of the flows receive a value *I(f) = 0*. This implies that either they have not been observed in malware at all (i.e., $P_M(f) = 0$), or they appear in all benign samples ($P_B(f) = 1$). The remaining 25% of the flows with non-zero weights can be grouped into two distinct classes: those with $0 < I(f) \leq 0.5$ (around 24%) and those with $0.5 < I(f) < 1$ (around 1%). Flows with $I(f) > 1$ are very rare and were observed mainly in malware samples only.

Fig. 3.4 shows the average and maximum flow weight values seen when grouping flows according to the Susi categories [159]. The distribution shows that, on average, flows with the highest weights are those related to unique identifiers (e.g., device and subscriber identities) and network information (e.g., hosts, ports and service providers) that end up being used in networking operations (e.g., connecting to specific URLs). The next most relevant weights belong to flows providing access to sensitive hardware information, including the subscriber ID and the SIM serial number, with sinks being methods send such data either via SMS or MMS. Table 3.4 contains some of the high-weighted information flows in terms of their $I(f)$ value. Overall, this provides an informative description of the behaviors (flows) observed in malware samples that do not appear in benign apps.

Source API methods from sensitive categories that appear in malicious flows (see Table 3.5) try to access sensitive unique identifiers, including DeviceID, SubscriberID, NetworkOperator and SimSerialNumber. Interestingly, sink API methods appearing in those flows often check if unique identifiers start with a given prefix (String.startsWith()),

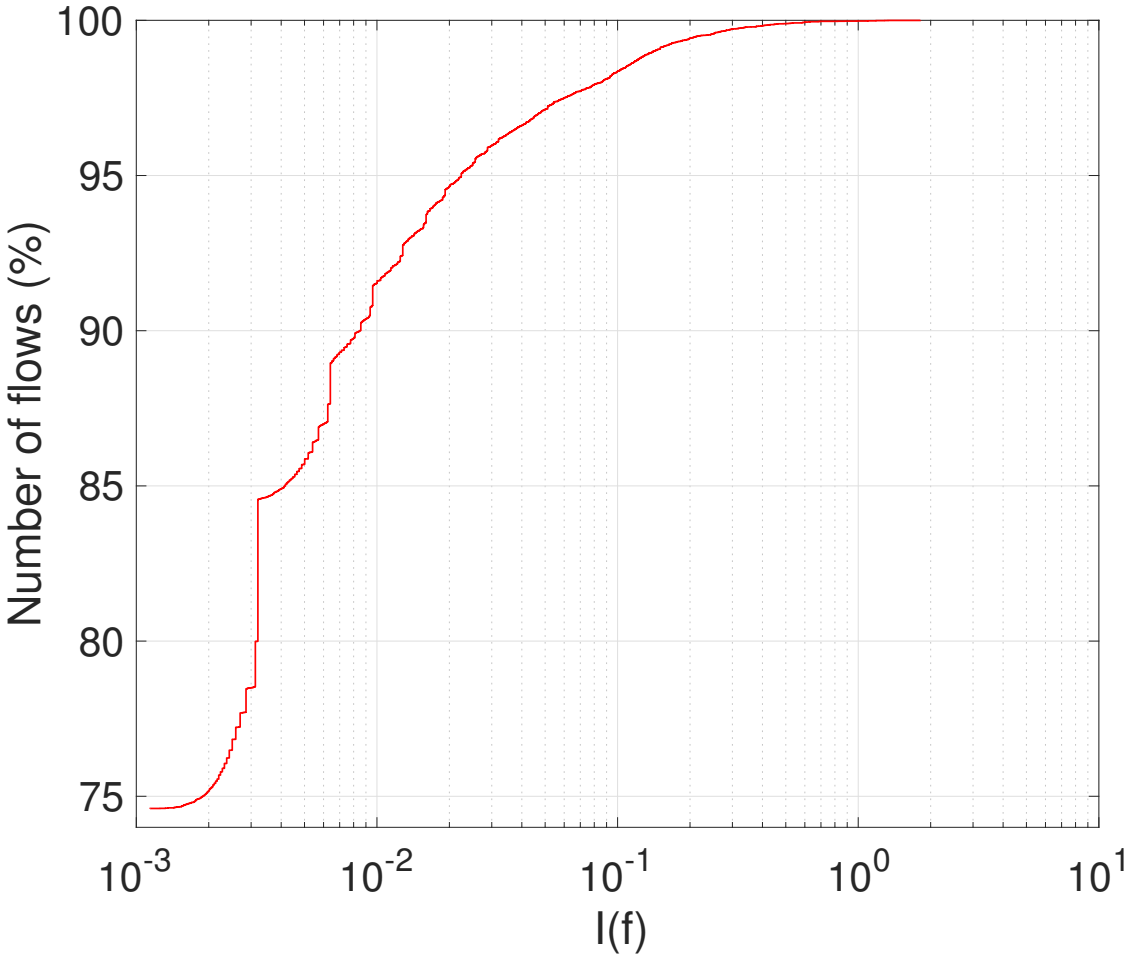


Figure 3.3: Cumulative probability distribution of the flow weight values $I(f)$. Note that the x-asis is given in logarithmic scale.

log them (Log.v()) or try to open a connection to a remote server (URL.openConnection()). Furthermore, source methods requesting the settings of the WebView class (WebView.getSettings()) which is used to display web pages or online contents within activities of an application or to design a new web browser and, also, the properties of the System class (System.getProperties()) which can be used to load files and libraries are popular in high-weighted flows. On the other hand, sink methods used to leak sensitive information through sending SMS messages (SM.sendTextMessage()) are also common in such flows.

After some preliminary experimentation, the distribution of flow weights forced us to slightly adjust the way the score is computed. The reason for this is that apps that contain a large number of information flows are penalized in their score since they accumulate a substantial number of tiny weights. To remove the effect of such tails, TRIFLOW implements

Table 3.4: Top ranked flows and their weight.

Source	Sink	$I(f)$
TM.getDeviceId()	String.startsWith()	0.69
TM.getDeviceId()	OutputStream.write()	0.26
TM.getDeviceId()	Intent.putExtra()	0.52
TM.getDeviceId()	String.substring()	0.28
TM.getDeviceId()	URL.openConnection()	0.37
TM.getSubscriberId()	String.startsWith()	0.88
TM.getSubscriberId()	OutputStream.write()	0.24
TM.getSubscriberId()	HttpURLCon.setRequestMethod()	0.25
TM.getSubscriberId()	URL.openConnection()	0.42
TM.getSubscriberId()	Intent.putExtra()	0.58
TM.getSimCountryIso()	Log.i()	0.37
TM.getSimCountryIso()	String.substring()	0.25
TM.getSimOperator()	Log.v()	0.31
TM.getNetworkOperator()	String.startsWith()	0.32
TM.getNetworkOperator()	String.substring()	1.18
TM.getLine1Number()	URL.openConnection()	0.20
TM.getLine1Number()	Log.v()	0.52
TM.getLine1Number()	String.startsWith()	0.53
TM.getSimSerialNumber()	String.startsWith()	0.98
TM.getSimSerialNumber()	String.substring()	1.09
gsm.SM.getDefault()	gsm.SM.sendTextMessage()	0.82
SM.getDefault()	SM.sendTextMessage()	1.81
NetworkInfo.getExtraInfo()	Log.d()	0.68
NetworkInfo.getExtraInfo()	String.startsWith()	0.45
WebView.getSettings()	WebS.setAllowFileAccess()	0.67
WebView.getSettings()	WebS.setGeolocationEnabled()	0.46
WebView.getSettings()	WebS.setPluginsEnabled()	0.50
System.getProperties()	String.substring()	0.45
PI.getBroadcast()	SM.sendTextMessage()	1.28
HashMap.get()	SM.sendTextMessage()	1.33

`TM`: TelephonyManager, `SM`: SmsManager, `PI`: PendingIntent,

`HttpURLCon`: HttpURLConnection, `WebS`: WebSettings.

two mutually exclusive strategies. The first one is simply to normalize the

MEAN

	LOG	FILE	NETWORK	SMS_MMS	AUDIO	NO_CATEGORY	LOCATION_INFORMATION
NETWORK_INFORMATION	0,0369	0,0074	0,0111	0,1767	N/A	0,044	N/A
CALENDAR_INFORMATION	0,0104	0,0096	0,0063	N/A	N/A	0,0148	N/A
LOCATION_INFORMATION	0,0342	N/A	0,031	0,0054	N/A	0,0173	N/A
DATABASE_INFORMATION	0,0277	0,0157	0,022	0,0655	0,0032	0,0179	N/A
ACCOUNT_INFORMATON	0,0027	N/A	N/A	N/A	N/A	0,032	N/A
UNIQUE_IDENTIFIER	0,0824	0,0079	0,3059	0,0919	N/A	0,0508	N/A
BLUETOOTH_INFORMATION	N/A	N/A	N/A	N/A	N/A	0,0031	N/A
NO_CATEGORY	0,0284	0,0173	0,0382	0,0799	0,0097	0,0222	0,0088

(a)

MAX

	LOG	FILE	NETWORK	SMS_MMS	AUDIO	NO_CATEGORY	LOCATION_INFORMATION
NETWORK_INFORMATION	0,684	0,0096	0,0257	1,8161	N/A	1,1881	N/A
CALENDAR_INFORMATION	0,0421	0,0128	0,0075	N/A	N/A	0,1284	N/A
LOCATION_INFORMATION	0,1403	N/A	0,1175	0,0128	N/A	0,1626	N/A
DATABASE_INFORMATION	0,2092	0,0544	0,0471	0,2336	0,0032	0,2766	N/A
ACCOUNT_INFORMATON	0,0028	N/A	N/A	N/A	N/A	0,1056	N/A
UNIQUE_IDENTIFIER	0,5216	0,0187	0,4279	0,1536	N/A	1,0901	N/A
BLUETOOTH_INFORMATION	N/A	N/A	N/A	N/A	N/A	0,0032	N/A
NO_CATEGORY	0,6032	0,4618	0,5292	1,3315	0,0376	1,1776	0,016

(b)

Figure 3.4: (a) Average and (b) maximum values of the flow weight distribution with flows grouped by SuSi categories (sources are placed in rows and sinks in columns). The group NO_CATEGORY refers to sources and sinks classified as non-sensitive in SuSi.

Table 3.5: Most relevant sources and sinks from sensitive categories.

Source Categories			Sink Categories			
NETWORK_INFORMATION	UNIQUE_IDENTIFIER	DATABASE_INFORMATION	LOG	FILE	NETWORK	SMS_MMS
getSerialNumber() getSubscriberId() getSimCountryIso() getNetworkCountryIso() getNetworkOperator() getAllMessagesFromSim() getWifiState() getHost() getRemotePort() getRemoteAddress() getLinkAddress() getNetworkPolicies() getDefault() getCellIdentity() getLatitude() getLongitude() getInstalledApplications() getAllPermissionGroups()	getDeviceId() getSimSerialNumber() getLine1Number() getSubscriberId() getNumber() getIccSerialNumber() getPhoneName() getServiceProviderName() getVoiceMailNumber() getAddress()	getConnectionId() query() getSyncState() getColumnNames() getColumnCount() getColumnIndex()	v() w() e() d() i() openFolder() startListening() storeFile() install() notify() setUserName()	write() dump() bind() setFileInput() openFileInput() openFileOutput() openDownloadedFile() sendto() readTextFile() sendfile() setOption() checkRead() checkWrite()	openConnection() setWifiApEnabled() selectNetwork() disableNetwork() setSerialNumber() setCountryCode() setNetworkPolicies() setMobileDataEnabled() setBandwidth() setHostname() setDeviceName() registerListener() setScanMode() processMessage() setAuthUserName() writeToParcel()	sendTextMessage() sendPdu() recordSound() sendData() sendDataMessage() setPremiumSmsPermission() dispatchMessage() append() disableCellBroadcast() moveMessageToFolder() setTextVisibility()

score by dividing the sum given in equation (3.2) by the number of flows in the app. This provides a fairer way of comparing apps of different size. The second approach consists of removing flows whose weight falls below a fixed cutoff value. In the remaining of this chapter, we will report results using the first strategy (i.e., score normalization), but our results suggest

that both perform equally well.

3.3.4 App Triage

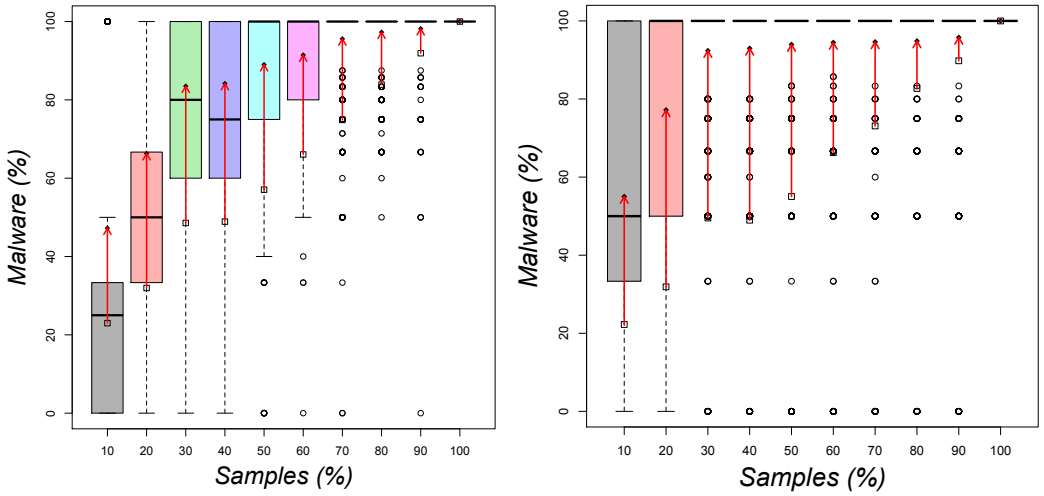
We next discuss the results obtained after scoring the apps in our dataset with the combined risk metric described in Section 3.2.1. As discussed before, such a risk score can be used to rank apps and prioritize analysis. In addition to this, TRIFLOW provides an explanation of the risk score similar to the one offered by Drebin [156] for the case of malware detection. In TRIFLOW, this consists of a break down of the score into the flows that contribute the most to it and a presentation to the user grouped by SuSi categories, which are generally easier to understand than the specific source-sink pair.

We compared TRIFLOW with other quantitative risk assessment metrics proposed in the literature. To do this we implemented various representative permission-based systems, including DroidRisk [162], Rarity Based Risk Score (RS) [158], and Rarity Based Risk Score with Scaling (RSS) [154]. As all these systems presented similar performance, in this section we only report results for RSS due to space limitations.

3.3.4.1 Scoring and Prioritizing Apps

Ideally, a triage system should maximize the time an analyst spends analyzing potentially harmful applications. Due to this reason, in this work we are primarily interested in reporting top ranked apps. Thus, we do not discuss the presence of other suspicious software such as *grayware* [33, 163] or obfuscated malware; we refer the reader to Section 3.4 for a more detailed discussion on this.

To quantify the performance of our triage system, we carried out the following experiment. We assume that the market operator only has time to manually vet a limited number of apps per unit of time (e.g., per day). We simulate a vetting process at different operational workloads w, ranging from 10% to 100% of the analyzed samples. More precisely, we assume that the operator receives batches of N samples per minute and their analysts are capable of processing 10%, 20%, ..., 100% of them. This constitutes a realistic scenario as some market operators can be more agile than others. The same applies to antivirus vendors. For instance, out of the 310,000 new samples received every day, Kaspersky Labs only processes



(a) RSS (Sarma et al. [154]). (b) TRIFLOW.

Figure 3.5: Results of the triage evaluation. Each plot shows the distribution of the fraction of malware correctly prioritized (y-axis) when a market operator can only afford to analyze $w\%$ of the samples (x-asis) at each time interval (e.g., daily-basis). Results are given for both RSS (left) and TRIFLOW (right). The red arrows within each plot represent the gain achieved by each scoring system with respect to a random prioritization policy.

1% manually (2 per minute)[2]. For our experiments we set $N = 10$, though the particular value is irrelevant for our analysis as it only constitutes a scale factor.

For each workload, we prepare a batch of samples containing randomly chosen samples from the joint goodware and malware datasets (recall that the malware-to-goodware ratio for testing is 1:5, so on average there will be 5 times more goodware than malware in each batch). Each sample in the batch is then scored and the top $w\%$ ranked samples are given to the analyst for a deeper analysis. We measure how many samples (in %) in that final block of samples passed on to the analyst are malware. We repeated this process 900 times, obtaining one percentage each time. For each workload w, the distribution of values is given in the boxplots shown in Fig. 3.5. We repeated the process for both TRIFLOW and RSS [154]. We also compared how both systems behave against a random ordering of the batch of samples. The square (□) symbol in each plot of Fig. 3.5 denotes the average ratio of malware samples given to the human analyst

[2]http://apt.securelist.com

after using a random prioritization policy, while the diamond (◇) symbol denotes the value given by the triage system. The red arrow joining them represents the difference between both numbers, i.e., the time saved after triaging the batch.

```
App: 4735ba2dfbdbb0f1e9a286da83155760c77dcce1bea9c4032ffd39792b251898.apk
Score = 2.78e-05
Score contributions:

1 [81.81 %] UNIQUE_IDENTIFIER -> LOG
  [ 0.03 %] 1.1 TelephonyManager.getDeviceId() -> Log.w()
  [ 0.05 %] 1.2 TelephonyManager.getSimSerialNumber() -> Log.w()
  [ 0.04 %] 1.3 TelephonyManager.getSubscriberId() -> Log.w()
  ...
2 [ 6.95 %] UNIQUE_IDENTIFIER -> SMS_MMS
  [ 6.95 %] 2.1 TelephonyManager.getSimSerialNumber() -> SmsManager.sendTextMessage()

3 [ 2.19 %] NETWORK_INFORMATION -> LOG
  ...
```

Figure 3.6: Snippet of a TRIFLOW report for a malware app belonging to the Plankton family.

Our results show that in all cases TRIFLOW can prioritize more malware samples per batch (see upper quartiles in Figure 3.5b) than RSS for every single workload value. Although not shown in the chapter, the results for DroidRisk [162] and RS [158] are similar. Remarkably, our approach performs better than RSS when the operators are overwhelmed. For example, TRIFLOW performs under a workload of 30% equally than RSS under a workload of 70%. Thus, the absolute number of malware samples analyzed after triage is 83% with RSS and 92% with TRIFLOW. When analyzing the overall improvement reported after a random triage (denoted with □ symbol), we can observe that TRIFLOW not only improves on average with respect to RSS, but also with respect to the most challenging cases (note that the distance between □ and the lower quartiles is notably larger in TRIFLOW). The same conclusions can be obtained by looking at the lower whiskers (worst cases without considering outliers), where a random triage perform surprisingly better than RSS for workloads from 10% to 60%.

3.3.4.2 Score Breakdown

TRIFLOW provides an informative break down of the score of an app in terms of each contributing information flow. This helps the analyst to understand why an app receives a particular score and how much each poten-

tial flow within the app contributes to it. The report is generated by sorting the flows predicted in the app in descending order of their $I(f)$ values and then computing how much (in %) they contribute to the total score. Fig. 3.6 shows an excerpt of a report describing a malware leaking sensitive information via SMS.

3.3.5 Efficiency

We now discuss the efficiency of TRIFLOW measured as the time required to obtain the score for an app. The scoring process has two main steps: extracting the sources and sinks of the app to construct the set $\hat{\mathcal{F}}$ of possible information flows, and then computing the score by adding up the product $\theta_f I(f)$ for each flow $f \in \hat{\mathcal{F}}$. The first step requires identifying all existing sources and sinks, whereas the second depends on the size of $\hat{\mathcal{F}}$, i.e., the number of sources times the number of sinks in the app. Fig. 3.7 shows both quantities for all the apps in our datasets (GooglePlay and Drebin). We consistently observe approximately twice the number of sources than sinks in each app, with an average of 290.10 and 176.81, respectively. The average size, measured as the number of potential flows, is 77,187.

Fig. 3.8 shows the overall time required to obtain the score for each app as a function of its number of potential flows. On average it takes 56 seconds to triage the entire app. The minimum and maximum scoring time for an app in our dataset is 0.01 seconds and 76.63 minutes, respectively. Approximately 50% of the apps require less than 31 s; 80% of the apps require less than 103 s; and 90% of the apps require less than 155 s. On average, TRIFLOW requires 2.3 ms per potential flow in the app. Execution times are not constant for a given size because not all potential flows will have a non-null probability of occurrence. The higher the number of flows with $\theta_f > 0$, the higher the number of risk terms that have to be added to the total score. This process is largely non-optimized in our prototype, hence the substantial variability observed in Fig. 3.8.

When processing a large dataset of apps, most of the computation time goes to the extraction of the information flows. Fig. 3.9 shows the comparison between the time taken by our approach and FlowDroid. We can observe that FlowDroid is computationally more intensive than TRIFLOW. In particular, we observe an improvement of about two orders of magnitude for smaller set of apps and about one order of magnitude for larger

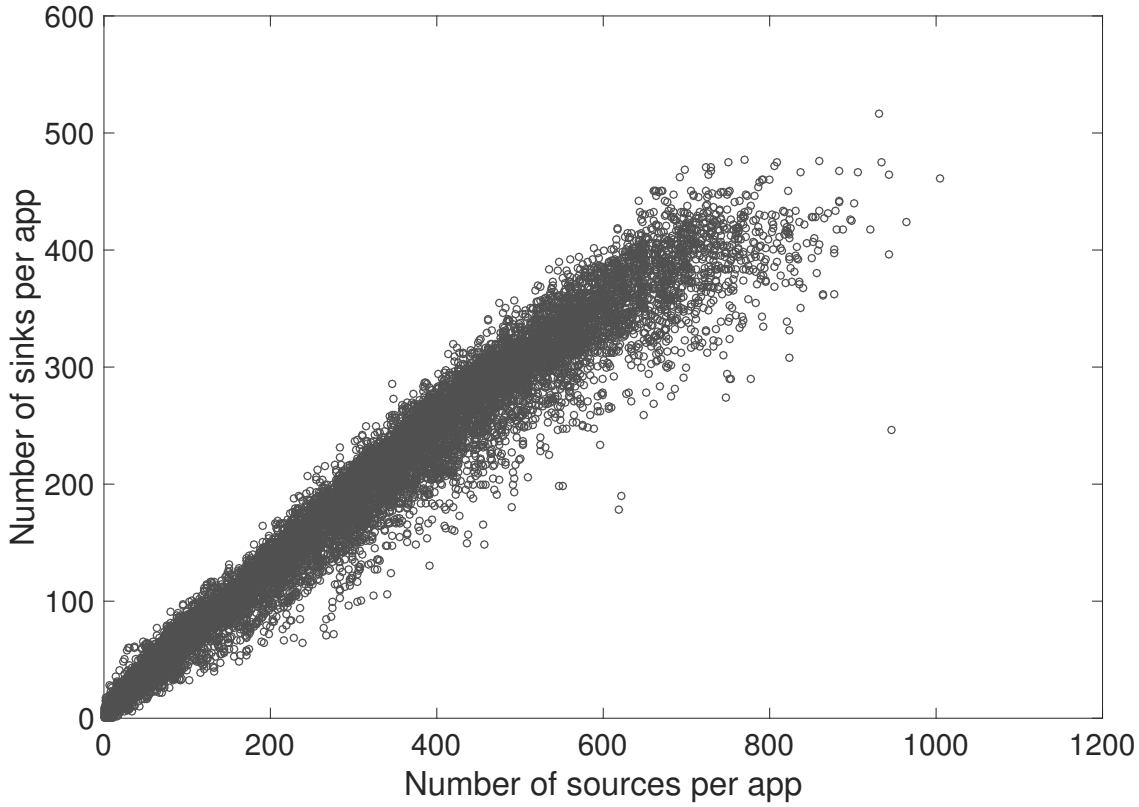


Figure 3.7: Number of sources vs number of sinks for all the apps in our datasets.

sets. This is a natural advantage of using a probabilistic predictor with respect to a precise tainting analysis, though it should only be used as an estimation for fast risk analysis.

3.4 Discussion

We next discuss a number of potential limitations of our approach related to its accuracy, the underlying risk notion, the validity of our results, and attacks against the scoring system.

3.4.1 Accuracy

A crucial step in TRIFLOW is the accurate identification of the sources and sinks present in an app. Our approach to do this is fast and robust (i.e., all sources and sink identified are actually in the app). It decompiles each app and looks into its smali code to find all sources and sinks. Still, it might not be accurate and in some cases, it might miss some sources or sinks.

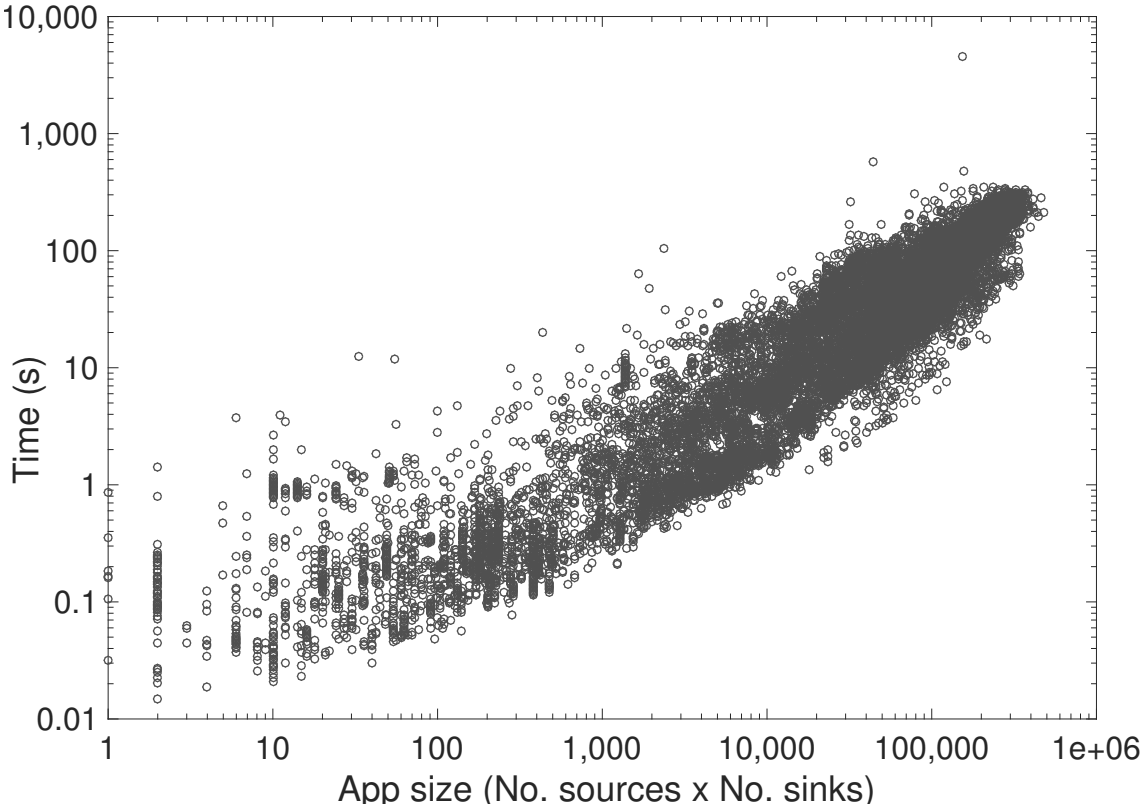


Figure 3.8: Scoring time for all the apps in our datasets as a function of each app's size measured as the total number of possible information flows. Note that the plot is in log-log scale.

The main cause for these inaccuracies is the use of reflection, particularly if methods are invoked dynamically at runtime. Since this cannot be determined at compilation time, such sources and sinks will certainly be missed by our approach. We do not know how much reflection is currently used by apps to access sources and sinks and, therefore, we cannot measure the extent of this limitation. However, apps leveraging reflection must use the java.lang.reflect package, so signaling this might provide the user with a warning about possible flows being missed by TRIFLOW.

3.4.2 Risk Notion

TRIFLOW scores apps according to the probable presence of *interesting* flows. In this chapter, we have quantified how significant a flow is using the mechanism described in Section 3.3.3, which captures how useful the flow might be to identify malicious apps. While we believe this is a useful risk metric, we also acknowledge that its use might easily lead to misin-

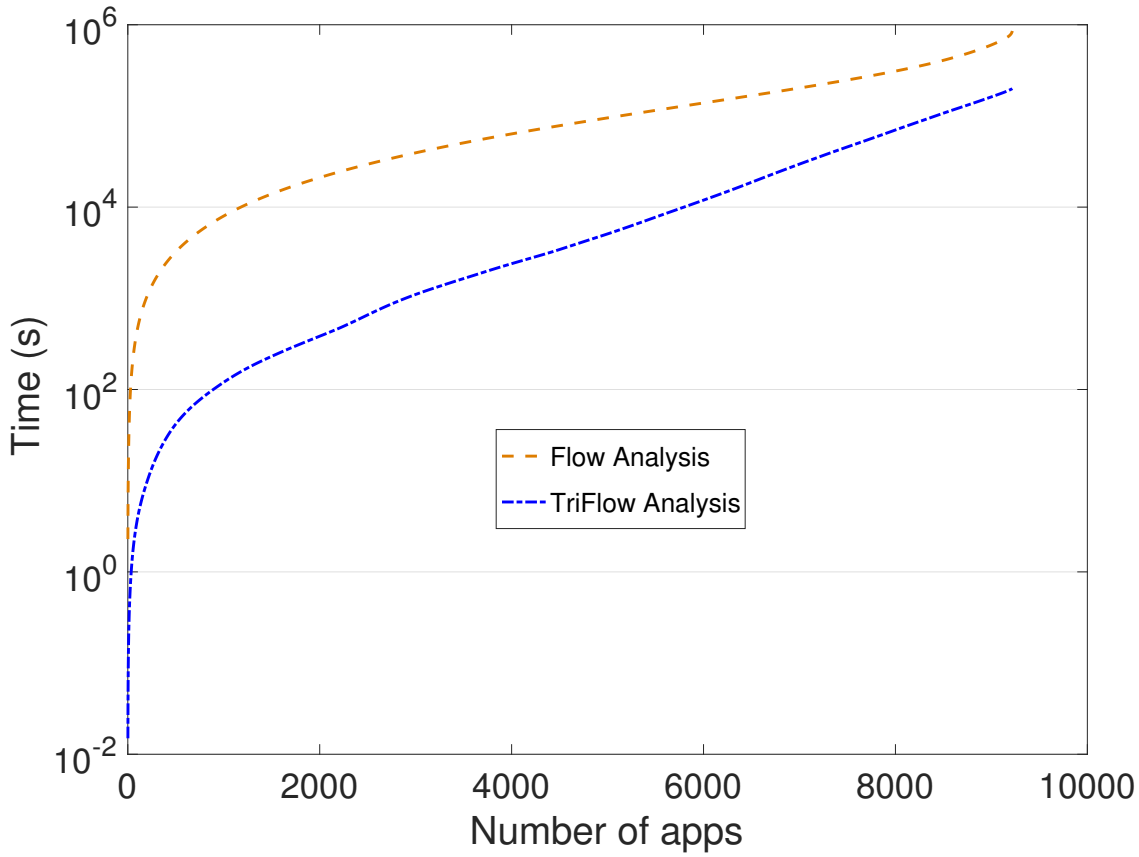


Figure 3.9: Cumulative time (in seconds) required to extract all possible information flows of a set of apps.

terpretations. Specifically, apps that score high should not be thought of as "likely malware," but simply as apps that possibly contain dangerous information flows (dangerous in the sense that are more frequent in malicious than in benign apps). During our experiments we came across some benign apps that score higher than many malicious samples, including, for instance, three known antivirus products (McAfee Mobile Security, NQ Mobile Security, and Vodafone Protect).

Our flow weighting scheme could be easily extended to incorporate other relevant flow features, or simply replaced by another measure of significance provided by the analyst (e.g., different weights for different SuSi categories). More generally, TRIFLOW should be viewed just as a risk metric finer-grained than permissions, and in a real setting its use should be complemented with other risk metrics that consider features of an app other than permissions or information flows.

3.4.3 Datasets

The experimental results discussed in this chapter might be affected by the number and representativeness of the apps in our datasets. While the exact coverage of our datasets cannot be known, we believe they are fairly representative in terms of different types of benign and malicious apps. For the latter we relied on the *Drebin* dataset, which extends the widely used *Malgenome* dataset and has been consistently used by most works in the Android malware area in the last two years. In the case of benign apps, we could only afford analyzing around 4000 applications, including 42 which are amongst the top most downloaded apps from Google Play in 2016. The limiting factor here is the extraction of information flows (with FLOWDROID, in our case), which requires a substantial amount of computational resources and, furthermore, fails for a large fraction of apps. This limitation is, in fact, one key motivation for our work. In any case, we did our best to avoid selection bias by choosing apps of different sizes and from different categories, prioritizing when possible those more popular (in terms of downloads) in the Google Play market.

3.4.4 Evasion Attacks

A sensible goal for an adversary is to modify his app so that it receives the lowest possible score. Since the score is monotonically increasing in the number of flows, adding sources or sinks will never decrease the score. To reduce the overall score an adversary will need to remove the use of some sources or sinks (which may affect the app's functionality), or just make them undetectable (e.g., as discussed above in the case of reflection). Alternatively, the adversary could try to replace current flows by others that use sources and sinks that are functionally equivalent to the original but receive a considerably lower weight. In our current implementation, this would only be possible by relying on methods rarely used by malware. We have not explored the extent of this limitation, and it is left for future work.

Our approach is vulnerable to collusion attacks since it does not consider information flows across apps (i.e., when the source is located in one app and the data is passed on to another app that access the sink). This can be seen as an extension to information flows of the classical permission

redelegation attacks [164], and can only be solved by extending individual analysis to groups of apps (e.g., such as in [165, 166]).

3.5 Related Work

This section presents and discusses some relevant works which are related to the topic discussed in this chapter, including information flow analysis in Android and permission-based risk metrics for Android apps.

3.5.1 Information Flow Analysis in Android

Information flows provide meaningful traces that describe how data components are propagated amongst the variables (and components) of a program [167]. Such flows can be used to represent the behavior of a given program, showing how and for what purpose programs are using specific pieces of information [157]. Any information flow is characterized by two main points defining the direction of the flow, known as the source and the sink. Sources are points within the program where sensitive data are obtained or stored in memory, while sinks are points where such data are leaked out of the program [168].

Unlike traditional desktop operating systems, apps in Android have their own life cycle and multiple execution entry points [169]. There are two types of information flows in Android applications. *Explicit* information flows analyze data-flow dependencies without considering the control-flow of the program. In contrast, *implicit* information flows analyze the control-flow dependencies between a source and a sink [170]. State-of-the-art analysis techniques (e.g., FlowDroid [92]) generally rely on explicit flows for two main reasons. First, implicit data flows can be tracked at a reasonable cost in most of the applications; and second, tracking such flows are unnecessary for many systems [170].

From another point of view, information flows are categorized as either *inter-app* or *intra-app* depending on the type of communication. Inter-app communication, and, as a result, inter-app information flows are established between components of two different applications [171, 172]. On the opposite side, intra-app data flows are those established between different components of the same application [173]. In addition, information flows are usually tracked using—*static* or *dynamic*—*taint analysis* [174].

Table 3.6: Information flow analysis tools for Android.

Tool	Type		Information Flows		Modeling Assumptions		
	Static	Dynamic	Explicit	Implicit	Callbacks	Life-Cycle	Native Code
FlowDroid [92]	✓		✓		✓	✓	✓
DroidSafe [93]	✓		✓		✓	✓	✓
CHEX [95]	✓		✓		✓		
LeakMiner [96]	✓		✓		✓		
AndroidLeaks [97]	✓		✓		✓		
TaintDroid [98]		✓	✓		✓	✓	✓
DroidScope [23]		✓	✓		✓	✓	✓

Static taint analysis aims at detecting privacy leaks before the execution of the application by constructing a control flow graph, while dynamic taint analysis tries to keep track of such leaks in run-time or in a customized execution environment [175].

There are several recent information flow analysis frameworks for Android (see Table 3.6). Static taint analysis tools such as FlowDroid [92], DroidSafe [93], FlowMine [94], CHEX [95], LeakMiner [96], and AndroidLeaks [97] have a relatively low run-time overhead with respect to other information flow frameworks. However, suffer from some critical issues that cannot be overlooked. On the one hand, they are imprecise as they need to simulate run-time behaviors [92], and, as a result, suffer from a high false positive rate [157]. On the other hand, some of these frameworks do not scale well with the number of applications [93]. Finally, applications could use advanced obfuscation techniques to hinder the extraction of information flows (e.g., [176]).

Similar to our approach, authors in MUDFLOW [157] use information flow analysis to study how malicious and benign apps treat sensitive data. MUDFLOW is able to establish a profile based on sensitive flows that allows them to characterize potential risks that are typically observed in malware. Our system, in a way, is motivated by these findings and by the fact that flow extraction involves a non-negligible amount of resources. In this chapter, instead of simply analyzing the abnormal usage of sensitive information, we use speculative information flows to further triage Android apps.

Dynamic taint analysis systems such as TaintDroid [98] and DroidScope [23] generally compensate for the lack of precision of static tools. However, these frameworks inherit the limitations of dynamic analysis systems, i.e., they may miss data flows from parts of the code not explicitly

exercised [93, 157]. Furthermore, apart from the fact that they impose a high run-time overhead [96], a malicious app could potentially fingerprint a given dynamic monitoring system to evade detection [92].

Tainting analysis frameworks are generally based on sensitive API calls tracking. Thus, it is paramount that this tracking considers the way apps interact with the system services. In Android, this interaction is stateless. This means that the taint analysis system has to take into account the life-cycle of applications and model all possible entry points and callbacks defined by the developer. Furthermore, sensitive API calls can also be declared in a native library outside of the main Dalvik Executable (DEX) and should also be modeled. Table 3.6 summarizes the most relevant information flow analysis frameworks discussed in each of the aforementioned categories together with the type of components modeled from the Android OS. Note that FlowDroid and DroidSafe are the only two static tainting frameworks that consider all modeling assumptions simultaneously.

3.5.2 Permission-Based Risk Metrics for Android Apps

The development of metrics and systems to assess risk in Android apps is an area that has received much attention in the last years. Works in this area have generally relied on metadata obtained from the app's package, such as requested permissions, and from the market, including the number of downloads, number of views, or the developer's reputation. Permission-based risk scores have been by far the most commonly explored because of two key advantages: permissions are relatively easy to understand by users and are compatible with the risk communication mechanism currently used in Android. Furthermore, app developers can reduce risk by avoiding the use of unnecessary permissions [158].

One of the seminal works in this area is [150], in which the authors propose a system based on a number of rules that represent risky permissions to flag apps. More recent contributions introducing permission-based risk metrics include DroidRanger [133], DroidRisk [162], MAST [177], WHYPER [178], RiskMon [152], MADAM [179], and the works of [180] and [181]. The risk metric proposed in DroidRisk [162] is based on the frequency and number of permissions an application request. In MAST [177], a risk signal is created based on the declared indicators of the app' functionality, such as permissions, intent filters, and the presence of native

code. The intuition behind this idea is that apps which are stronger in terms of finding relations between these indicators impose a higher magnitude of risk and, thus, should be flagged as malicious. WHYPER [178] uses natural language processing techniques to reveal why an app may need a specific permission, paying attention to permissions' purposes. MADAM [179] relies mainly on metadata from the market, including the developer's reputation and market provenance. Finally, RiskMon and the work in [181] consider API traces as well, since some of them are critical and do not require any permissions. Finally, [158], [154], and [153] assign high risk scores to permissions or combination of permissions that are critical and rarely requested by the apps in the same category.

As permission-based metrics are based on metadata of the app obtained through static analysis, they can be imprecise and prone to errors. Other metrics have tried to overcome this by looking into features other than permissions. For instance, RiskRanker [151] introduces a risk signal based on root exploits, while [181] proposes a risk score considering static metadata, dynamic information from intents, components, network usage, and the app's behavior (e.g., whether an app launches other apps). Finally, the majority of metrics, except [152] and [180], do not take into account the security requirements or expectations of smartphone users. This is particularly important in practice, since risk ultimately depends on each user's preferences and execution context.

Our approach is complementary to most of these works. While we share the goal of quantifying risk, our primary focus is not on malware detection, but on prioritizing information flow analysis. Furthermore, our flow-based scoring mechanism can be easily integrated with existing metrics based on other risk factors to provide a more comprehensive risk assessment.

3.6 Conclusion

In this chapter, we designed and implemented a novel tool, called TRIFLOW, that automatically scores Android apps based on a forecast of their information flows and their associated risk. Our approach relies on a probabilistic model for information flows and a measure of how significant each flow is. Both items are experimentally obtained from a dataset containing benign and malicious apps. After this training phase, the models are used

by a fast mechanism to triage apps, thus providing a queuing discipline for the pool of apps waiting for a precise information flow analysis.

Our experimental results suggest that TRIFLOW provides a sensible ordering based on the potential interest of the app. Given the huge amount of computational resources demanded by information flow analysis tools, we believe this could be very helpful to maximize the expected utility when dealing with large pools of apps. Additionally, TRIFLOW could also be used as a standalone risk metric for Android apps, providing a complementary perspective to alternative risk assessment approaches based on permissions and other static features. Finally, to encourage further research in this area, we make our results and implementation available online.

4

Behavioral labeling of Android malware families

4.1 Approach overview

The popularity of the Android operating system and its openness have led to a significant rise in mobile malware targeting this platform in recent years [182] [183]. In this context, several works have addressed the problem of how to automatically characterize and label Android malware. This pursues to main goals: to obtain a clearer understanding of the threat landscape in this area, and to assist malware detection, since this is generally not possible without having a clear understanding of how the malicious program behaves.

Despite its importance, malware labeling is not an easy task. The majority of labels (also known as families [15]) assigned to malware samples by AntiVirus (AV) engines are not consistent with their real behavior [15]. One of the main causes for these inconsistencies is the lack of appropriate standards for naming malware across different vendors [16]. In most cases, these labels are assigned to malicious apps based on static information, including data about the developer, the source country, code structures [17], etc. While these features could be obtained quickly, they might be imprecise as they do not reflect how malware interacts with the victim device and data. Moreover, they can be modified simply to bypass the labeling system [18].

Due to the current inconsistencies which exists between malware family labels and their real behaviors, several variants (sub-families) of apps have recently been identified in different families [79, 184]. In addition, often there is not a clear and meaningful separation between malware families in terms of their behavioral profiles in various malware datasets. Many apps are classified into one family even if they exhibit behaviors that could be

attributed to various families. To overcome these limitations, in this work we examine how to associate with each app (and family) a profile that can be used to characterize its behavior.

The primary goal of this work is to explore how the family labeling of popular Android malware is not always aligned with the actual behavior of such apps. Our main observation is that apps in the same family can behave differently; and, conversely, apps in different families can exhibit almost identical behaviors. Although this may also happen in benign apps, the focus of this work is on malware.

Our approach to characterize malicious apps is based on the set of all real information flows extracted from each application using a precise static taint analysis tool. Information flows are the basis of our characterization scheme as they can potentially reveal the behavior of applications. Generally, each information flow, denoted by $f = (s, k)$, is consisted of two main points known as source (s) and sink (k). Sources are calls to API methods which can be used to obtain sensitive information from the user or the system, while sinks are calls to API methods which can leak such sensitive data in different ways, such as logging those sensitive data, and, then, sending them through a text message to a remote server.

Although flows are interesting and fine grained, they may provide limited information about the apps' behavior. For instance, an application may contain a sensitive flow logging our contacts into a file. However, this flow cannot represent the general intention of the app unless it is analyzed at a higher level, i.e. what other combinations of flows are used with this sensitive flow. If this app sends these saved contacts to a remote server, it will have a different behavior, and will thus impose a different amount of risk comparing with the same app sending SMS advertisements to all the logged contacts.

To overcome these limitations, our characterization scheme considers a larger scale which is patterns of flows, $P =< f_1, f_2, ..., f_N >$; or, in other words, different combinations of information flows observed in each application. Comparing with flows, patterns are even more interesting as they may be seen as behavioral building blocks for applications. To achieve this, we first extract combinations of flows that appear in each sample as this can help to identify the real behavior of apps and the way they are treating sensitive user or device information. To study whether or not each family can be associated with a set of flow patterns, we leverage a well-known text

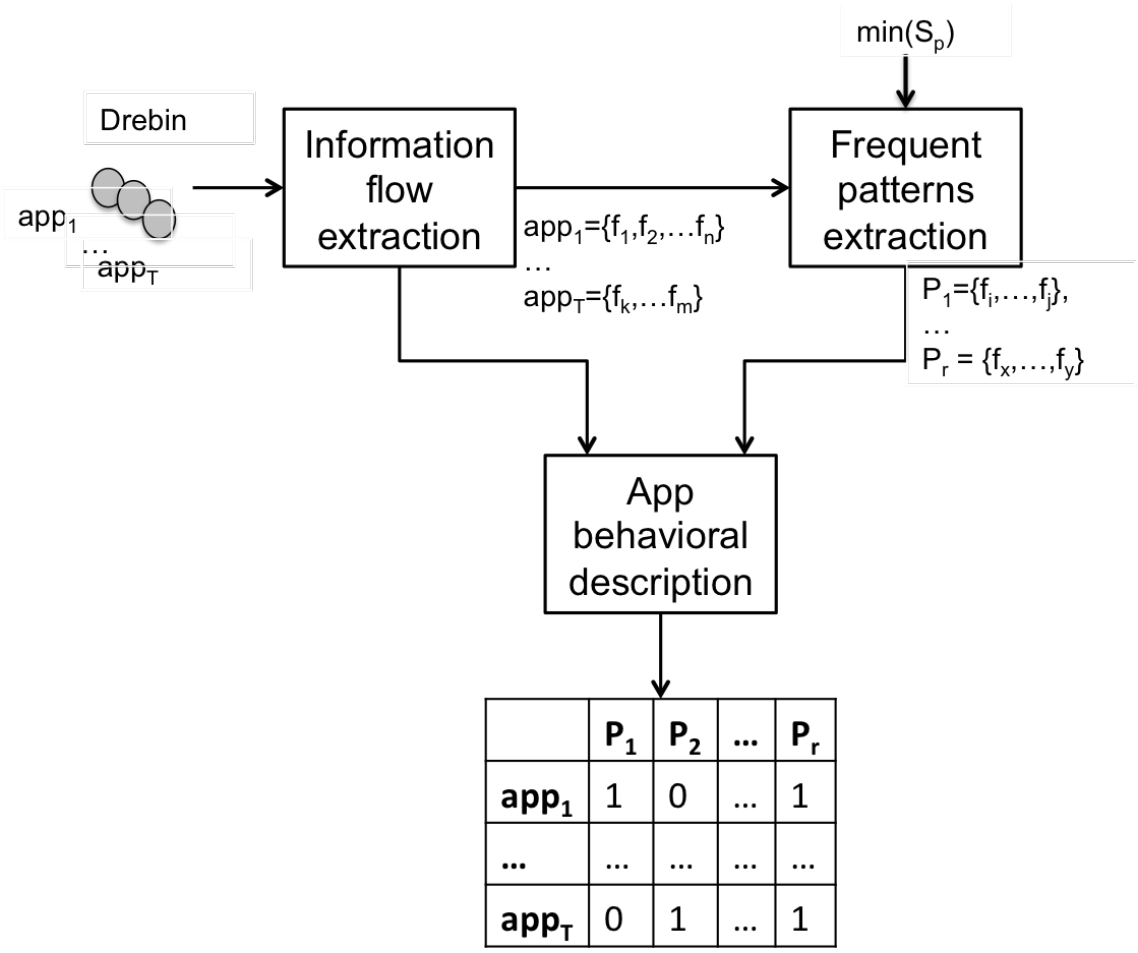


Figure 4.1: Behavioral analysis procedure

mining and information retrieval technique and analyze behavioral similarities among families (See Fig. 4.1).

Finally, we use cluster analysis to group the apps in our datasets according to their behavioral profiles. We provide insights of the semantics of each cluster by analyzing the flows of their apps in terms of SuSi categories [2], which provide a convenient way of summarizing the behavioral profiles. Our results validate our initial hypothesis, and we show examples of both: (1) apps in the same family that behave differently; and (2) apps in different families that behave identically.

The rest of this chapter is organized as follows. Section 4.2 introduces our analysis of information flow patterns in Android malware and how they relate to family labels. Section 4.3 describes our approach to behavioral clustering for apps using patterns of information flows and discusses the experimental results obtained. Section 4.4 discusses potential limitations of our approach and threats to validity, and Section 4.5 describes related

work in this area. Finally, Section 4.6 concludes the chapter and proposes some future works.

4.2 Frequent Information Flow Patterns in Malware Families

In this section we discuss our analysis of information flow patterns in Android malware. We first describe the datasets we used in our work. Then, we discuss the procedure used to extract information flows and the results we obtained. Finally, we extract and analyze frequent patterns of information flows (i.e. groups of flows that frequently appear together) and how they correlate with family labels.

4.2.1 Datasets

We used two different datasets to carry out the experiments; a frequently used Android malware dataset, known as Drebin [76], and a recently released one, called AMD [79].

The Drebin dataset contains 5,560 malware samples from 179 different families collected between 2010 and 2012. The AMD dataset, released in 2017, consists of 24,650 apps from 71 families discovered between 2010 and 2016. In the AMD dataset, each family is further divided into subfamilies (or variants) based on different criteria, such as the way in which apps are composed, installed and activated, to name a few. It contains 135 Android malware variants in total and covers a wide range of types of malware, including trojans, ransomware and adware.

Furthermore, the family intersection of these two datasets is shown in Fig. 4.2. As it is clear, there are 18 families and 739 apps in common between these two datasets. From this amount, around 65% and 17% of apps are from two popular Android malware families, DroidKungFu and FakeInst. These two families are later divided into 6 and 5 different subfamilies in AMD dataset respectively.

4.2.2 Extracting Information Flows

We used FLOWDROID [92] to identify data flows in all apps of both datasets. FLOWDROID is a static taint analysis tool that takes into ac-

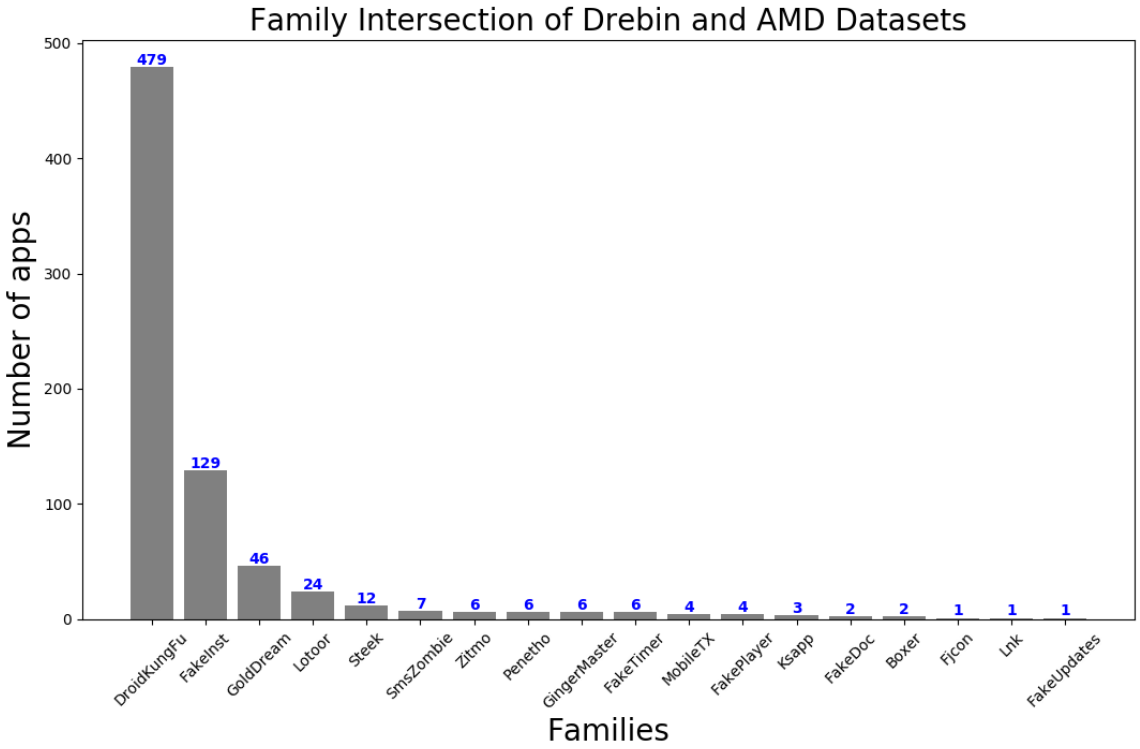


Figure 4.2: The intersection of AMD and Drebin datasets.

count the life cycle of an Android app, its components and its UI widgets. We ran FLOWDROID considering all Android API sources and sinks proposed in the SuSi project [2]. The extraction took place on a 2.6 GHz Intel Xeon Ubuntu server with 40 processors and 200 GB of RAM. We set a timeout of 30 minutes and assigned between 40 GB and 100 GB of RAM per app. Even with this configuration, FLOWDROID could not finish the flow extraction process entirely for all the apps in our datasets[1]. It could successfully analyze 4,648 and 18,811 apps from Drebin and the AMD datasets, respectively. To guarantee that the final dataset reflects the family distribution of the original one, we sampled the results and retained 4,361 apps from Drebin and 14,297 apps from the AMD dataset.

In total, we extracted 16,634 flows from Drebin and 585,968 from the AMD dataset. The distribution of flows across families has a high variability, ranging from very few (just 1 or 2) to a maximum of 2,791 in the case of DroidKungFu. The average number of flows per app is 26 in the Drebin dataset. Eighty eight families had less than 100 unique flows; 9 families had between 100 and 200 flows; and the remaining families had more than

[1]This low reliability has been reported before (see, e.g., [157, 185]) and is indicative of the limitations of current techniques to extract information flows.

200 unique flows. In the AMD dataset, each application had around 41 unique information flows on average.

4.2.3 Frequent Patterns of Information Flows

We used the set of information flows extracted from our datasets to identify common patterns appearing in the apps that belong to the same family. To do this, we applied a popular and fast frequent itemset mining algorithm known as FPClose [186]. In our case, an itemset is a set of flows with a specific support value. Note that this is an unordered set of flows which cannot always be serializable, i.e., it is generally not possible to establish which flow happens before or after another one. The support of an itemset measures how many apps in the family contain such pattern (i.e., all flows in the set). An itemset is called *frequent* when its support is greater than or equal to some given threshold $min(S_p)$. A frequent closed itemset is a frequent itemset that cannot be found in any proper superset having exactly the same support. This is exactly what we look for when characterizing frequent itemsets of flows for each family, hence our choice of the FPClose algorithm.

We relied on SPMF [187], an open-source data mining library specialized in pattern mining, to identify frequent closed itemsets. We set $min(S_p) = 0$, which allows us to extract *all* patterns within each family regardless of their support. Since our focus is on analyzing the relationships between samples belonging to the same family, we discarded those families with only one app. This left us with a reduced dataset of 4,204 apps from 115 different families for the Drebin dataset. The AMD dataset was not affected as all families have more than one sample. The FPClose algorithm extracted 101,349 and 1,578,364 unique flow patterns from Drebin and AMD, respectively.

The size of a pattern P, denoted $size(P)$, is the number of flows it contains. Fig. 4.3 shows the distribution of pattern sizes in the Drebin dataset. Pattern sizes range from 1 to 467 flows, with an average of 120, though the distribution is multi-modal. The majority of patterns (32.91%) have between 150 and 170 flows, while very few of them (0.3%) have more than 190 flows. The longest pattern includes 467 flows, and 252 patterns (0.2%) contain just one information flow. In the AMD dataset, pattern sizes range from 2 to 879 flows with an average of 64 approximately.

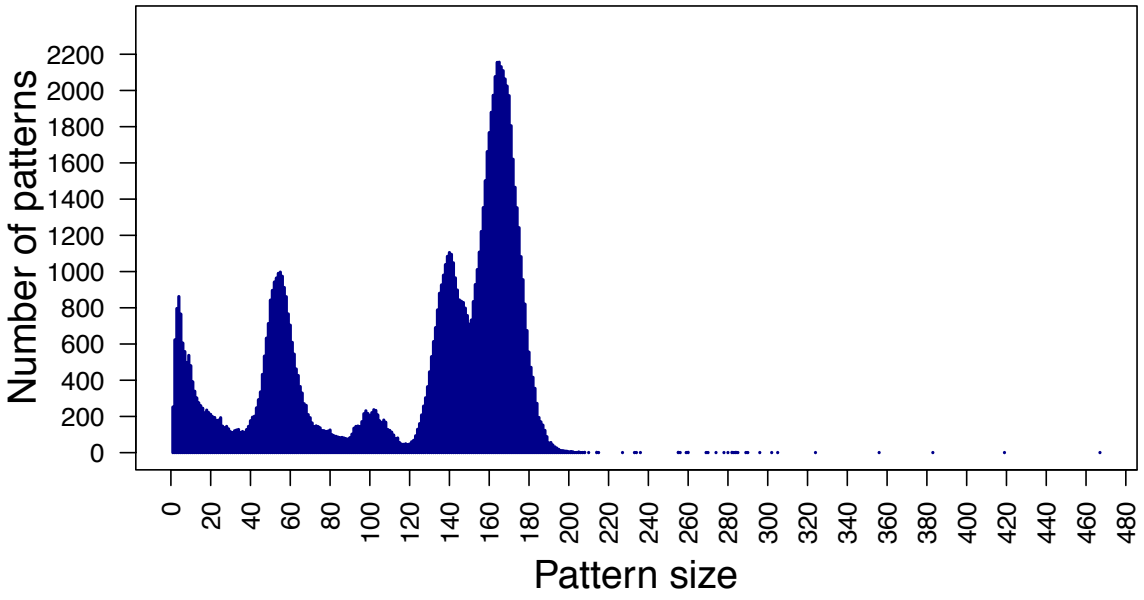


Figure 4.3: Distribution of pattern sizes (Drebin dataset).

The distribution of support values (again, for the Drebin dataset) is shown in Fig. 4.4. Note that the support is given in absolute values (i.e., the number of apps in which the pattern is observed), and not as a percentage. Specifically, 13.94% of patterns have a support in the range $[15, 25]$; 67.88% of all patterns have support in the range $[5, 15)$; and 12.80% of patterns have a support value less than 5. In AMD, 11% of patterns have support in the range [15, 25) and 24% of patterns have support in range of [5, 15); and, 3% of patterns have a support value less than 5. The key finding here is that the majority of patterns are associated with very few apps. It is unclear, however, if the sets of apps that can be characterized by the same flow patterns belong, or not, to the same malware family. In other words, do a particular set of flow patterns characterize with sufficient precision the apps belonging to the same malware family? We explore this question in the next section.

4.2.4 Malware Family Classification Using Flow Patterns

We now aim at studying whether each malware family in the two datasets can be associated with a distinct set of patterns that represent the behavior of the apps in the family. To do this, we leveraged a classic technique used in text mining and information retrieval to quantify the utility of query words to characterize documents: the so-called *TF-IDF (Term Frequency-*

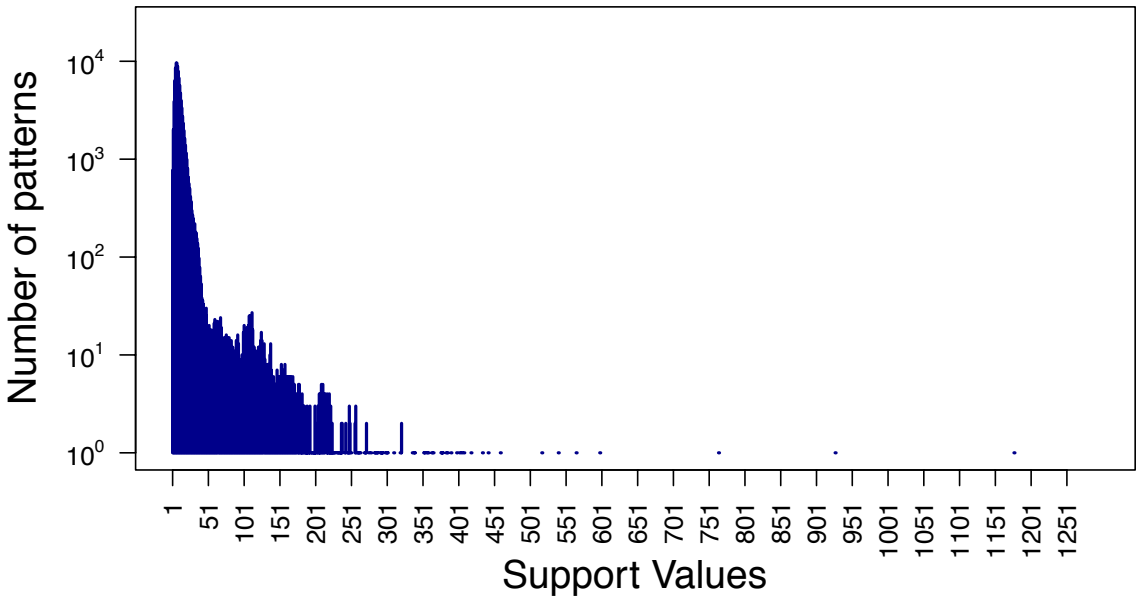


Figure 4.4: Distribution of patterns support values (in logarithmic scale) in Drebin dataset.

Inverse Document Frequency) weighting scheme. The intuition behind it is very simple: the utility of a term (word) to characterize a document is proportional to how frequent the term is in the document, and inversely proportional to how common the term is across documents. The TF-IDF scheme has been widely used in domains other that text mining, including as a step to do feature selection in malware classification [17, 124, 188].

We calculated the TF-IDF value for each pattern in our datasets considering each malware family as the *retrieval unit*. Thus, the TF term of a pattern P in a family F is obtained as

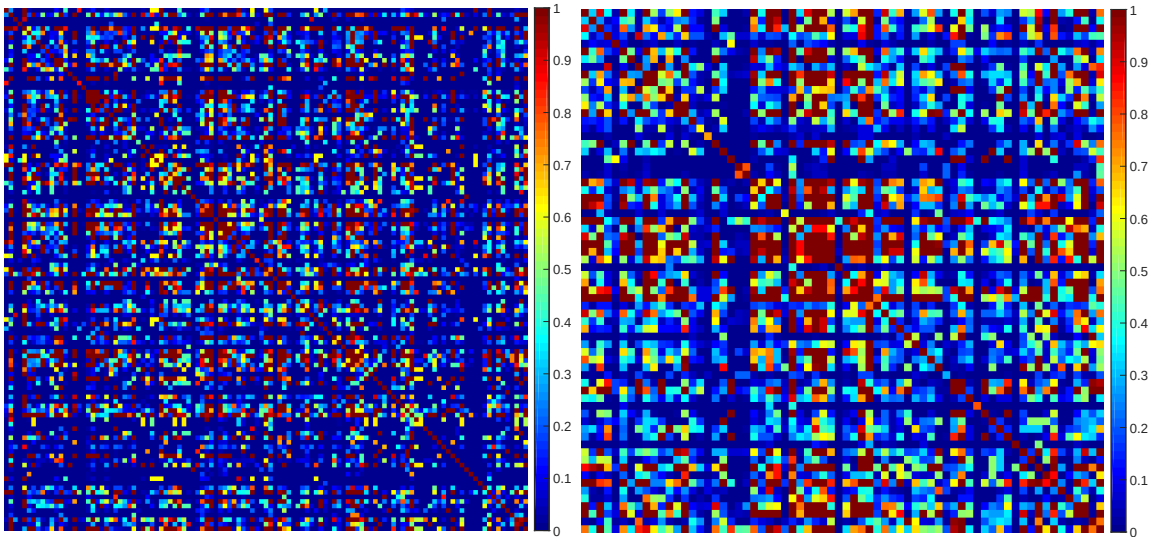
$$TF(P,F) = \frac{\text{No. times } P \text{ appears in } F}{\text{No. patterns in } F}. \tag{4.1}$$

For the IDF term, we used the standard definition

$$IDF(P) = \log \frac{\text{No. families}}{\text{No. families with pattern } P} \tag{4.2}$$

Both terms are then used to measure the relationship between each pattern P and each family F as

$$TF - IDF(P,F) = TF(P,F)IDF(P). \tag{4.3}$$



(a) Drebin dataset (b) AMD dataset

Figure 4.5: Similarity matrices between malware families using the cosine similarity between the TF-IDF vectors associated with information flow patterns. Each row and column in the matrix represents a family. Families have been arranged in the same order from left to right and from top to down, hence the maximum similarity observed along the main diagonal. Family labels have been removed for better readability.

the result is a set of vectors $\{T_F\}$, one per family F, in which each component is the TF-IDF value associated with a particular pattern. Thus, a pattern with high TF-IDF for a particular family is highly relevant in characterizing that family, and vice versa.

4.2.5 Behavioral Similarity Among Families

Figs. 4.5a and 4.5b show the similarity between every possible pair of families in both datasets. The similarity between two families is computed using the standard cosine distance between their TF-IDF vectors. Thus, if X and Y are the TF-IDF vectors of two families, their similarity is given by the cosine similarity metric

$$sim(X,Y) = \frac{X \cdot Y}{\|X\|_2 \|Y\|_2} = \frac{\sum_{i=1}^{n} X_i Y_i}{\sqrt{\sum_{i=1}^{n} X_i^2}\sqrt{\sum_{i=1}^{n} Y_i^2}}. \tag{4.4}$$

The results obtained reveal high similarity among a large number of families that are, in principle, unrelated. They also supports the idea that

(patterns of) information flows can be seen as building blocks to characterize family behavior, and are useful to detect similarities and differences across apps and families.

Raw information flows (as opposed to flow patterns) could also be used as feature vectors to characterize each family. This has been previously explored to, for example, classify malware samples into families [189]. However, our experiments in this regard discourage this approach. Using flows as features increases the granularity of the described behavior, which affects negatively the similarity between apps–and, therefore, families. In particular, it results in similarity matrices in which too many apps in different families look too similar to each other mainly because they share a limited number of flows. Instead, considering patterns of flows as features translates into a characterization in which apps are similar only if they share sets of flows, some of them quite large, that appear together.

4.2.6 Classifying Apps into Families

The TF-IDF modeling approach also provides an implicit procedure to classify apps into families. Each app is associated a binary feature vector in which the j-th component is set to 1 if the flow pattern P_j appears in the app, and 0 otherwise. The similarity between the app and each family F is computed using the cosine metric between both vectors, and the family with the highest similarity is chosen.

We used this procedure to obtain the predicted family for all apps in both datasets. The average classification errors are 18.50% and 21.48% for Drebin and AMD, respectively. However, errors are not evenly distributed across families. For example, for apps in 69 families from Drebin (60%) and 44 families from AMD (64.70%), the classification error is 0. Others exhibit errors around 1%, while for some families it is extremely large (91%). These results reinforce the preliminary conclusions reached before: in general, there is not a one-to-one correspondence between family labels and behavior (expressed in terms of information flow patterns). This means that some families contain apps that behave differently, and also that there are apps that behave almost identically despite belonging to different families. Both cases are the source of classification errors when behavioral features such as information flow patterns are used to characterize apps.

Two natural questions are: how many distinct behavioral classes are there and how they correlate with the family labels? We explore both points in the next section.

4.3 Behavioral-Based Malware Clustering

In the preceding sections, it has been shown that malware family labels are not relevant to describe the actual behavior of their member apps. However, it remains unclear whether or not malware samples have similarities in their behavior. If it is so, it would be possible to cluster them according to their behavior, offering groups of samples that behave similarly irrespective of their family. This section explores this idea.

Choosing an appropriate set of features and an efficient clustering algorithm are both critical in clustering malware samples. These issues are addressed in Sections 4.3.1 and 4.3.3, respectively. Afterwards, the clustering results and the comparison with family labels is discussed in Section 4.3.4.

4.3.1 Behavioral Features

As discussed in the preceding section for malware families, we characterize the behavior of each sample using its information flow patterns. Each sample is thus associated with a binary vector in which the j-th component is set to 1 if the sample contains pattern P_j, and 0 otherwise. Our choice of flow patterns instead of raw flows intends to reduce the dimensionality and sparsity of the feature vectors. Even with the use of flow patterns, this leads to vectors with 101 K and 1.5 M items per sample (see Section 4.2.3) for Drebin and AMD, respectively. In addition, interpreting the behavior encoded in such vectors would be hard, which will affect negatively the usability of the resulting system.

To overcome this limitation, we replaced each flow by its SuSi category [2]. Each category contains particular type of sensitive API calls (e.g., calls to extract the geographical location), which are helpful to understand the underlying behavior omitting the specific API call used to achieve such behavior. We consider a standard set of 17 source and 19 sink categories (see Table 4.1), which results in 323 possible combinations of source and sink categories. Note that *NO_CATEGORY* is both a source and a sink

category containing flows that are not identified as sensitive. Therefore, the behavior of each sample is represented by a binary vector of 323 features.

Table 4.1: Source and sink categories in SuSi ([2])

Source Categories	Sink Categories
UNIQUE_IDENTIFIER	*LOCATION_INFORMATION*
LOCATION_INFORMATION	*PHONE_CONNECTION*
NETWORK_INFORMATION	*VOIP*
ACCOUNT_INFORMATION	*PHONE_STATE*
FILE_INFORMATION	*EMAIL*
BLUETOOTH_INFORMATION	*BLUETOOTH*
DATABASE_INFORMATION	*ACCOUNT_SETTINGS*
EMAIL	*AUDIO*
SYNCHRONIZATION_DATA	*SYNCHRONIZATION_DATA*
SMS_MMS	*NETWORK*
CONTACT_INFORMATION	*FILE*
CALENDAR_INFORMATION	*LOG*
SYSTEM_SETTINGS	*SMS_MMS*
IMAGE	*CONTACT_INFORMATION*
BROWSER_INFORMATION	*CALENDAR_INFORMATION*
NFC	*SYSTEM_SETTINGS*
NO_CATEGORY	*BROWSER_INFORMATION*
	NFC
	NO_CATEGORY

Tables 4.2 and 4.3 show the distribution of flows for both datasets. Most malware specimens have source methods that access sensitive information, including location, network, databases and unique identifiers. These are then leaked through various channels, most notably through text messages or directly through the network. We observed 6,040 flows equivalent to saving critical network information in JSON files (e.g., *WifiInfo.getMacAddress()* $\rightarrow$ *JSONObject.put()*); 5,349 flows equivalent to saving location information in JSON files (e.g., *LocationManager.getCellLocation()* $\rightarrow$ *JSONObject.put()*); 2,598 flows equivalent to saving unique identifiers into JSON files (e.g., *TelephonyManager.getSimSerialNumber()* $\rightarrow$ *JSONObject.put()*); and, finally, 1,117 flows equivalent to saving database information in JSON files (e.g., *SQLiteQueryBuilder.query()* $\rightarrow$ *JSONObject.put()*). Malware gen-

erally sends such information, saved in JSON format, over the network (2,980 flows such as *JSONObject.getJSONObject()* → *HttpURLConnection.setRequestMethod()*) than through text messages (884 flows like *JSONObject.getString()* → *SmsManager.sendTextMessage()*). Some malware families of banking Trojans have extra capabilities over the apps in other families. For instance, variants of the BankBot malware (≈ 30%) have the ability to record sound and log keystrokes as well as leaking sensitive information through the network.

Table 4.2: Number and distribution of flows extracted from the applications in the Drebin dataset grouped by SuSi categories. Only flows whose contribution is greater than 1% of the total are shown.

Source Category	Sink Category	#Flows	Contribution (≈%)
NO_CATEGORY	*NO_CATEGORY*	10,612	63.80
NO_CATEGORY	*LOG*	1,996	12.00
NETWORK_INFORMATION	*NO_CATEGORY*	778	4.68
UNIQUE_IDENTIFIER	*NO_CATEGORY*	743	4.47
NO_CATEGORY	*NETWORK*	406	2.44
NO_CATEGORY	*SMS_MMS*	308	1.85
NETWORK_INFORMATION	*SMS_MMS*	289	1.74
DATABASE_INFORMATION	*NO_CATEGORY*	269	1.62
LOCATION_INFORMATION	*NO_CATEGORY*	253	1.52
NETWORK_INFORMATION	*LOG*	193	1.16

Table 4.3: Number and distribution of flows extracted from the applications in the AMD dataset grouped by SuSi categories. Only flows whose contribution is greater than 1% of the total are shown.

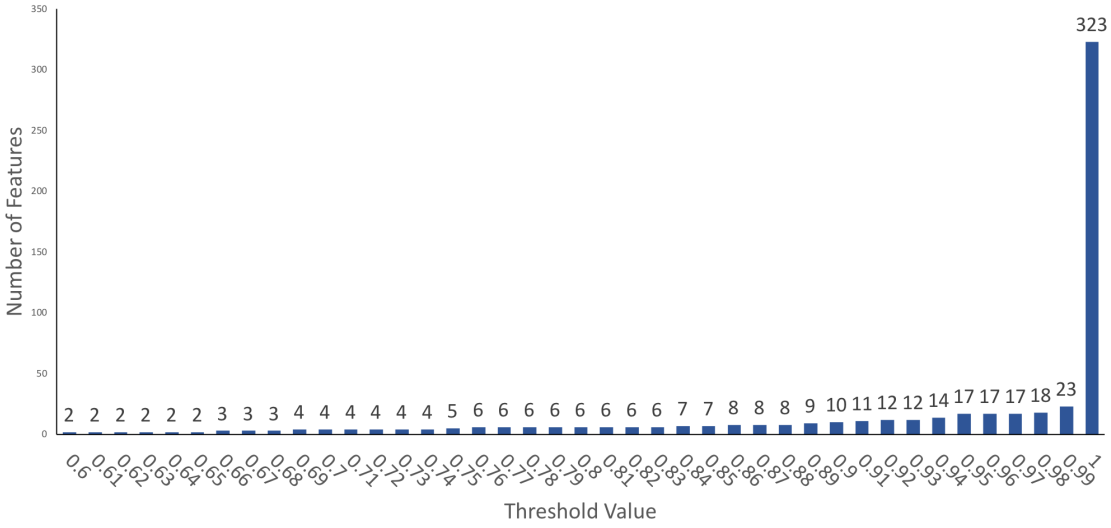
Source Category	Sink Category	#Flows	Contribution (≈%)
NO_CATEGORY	*NO_CATEGORY*	395,635	67.51
NO_CATEGORY	*LOG*	89,187	15.22
NETWORK_INFORMATION	*NO_CATEGORY*	23,371	3.98
UNIQUE_IDENTIFIER	*NO_CATEGORY*	15,238	2.60
LOCATION_INFORMATION	*NO_CATEGORY*	13,408	2.28
DATABASE_INFORMATION	*NO_CATEGORY*	9,126	1.55
NO_CATEGORY	*NETWORK*	6,909	1.17

4.3.2 Feature Selection

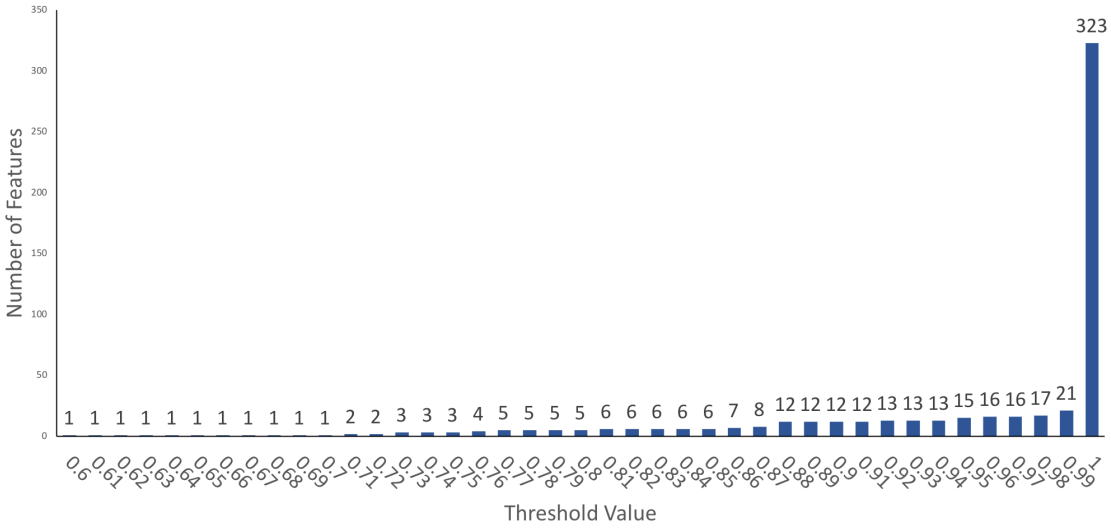
After inspecting the distribution of the values taken by some features across all samples, it became clear that some of them had a prevalence of either ones or zeros. Thus, they offered little help in highlighting behavioral differences between samples. We experimentally determined a threshold percentage of prevalence among all samples of 0.99, thus intending to keep as many relevant features as possible. Figures 4.6a and 4.6b show the amount of remaining features after removing those exceeding the prevalence threshold in Drebin and AMD datasets. Based on this analysis, we set the threshold to 0.99 so as to keep as many relevant features as possible. This left us with the 23 features for the apps in Drebin dataset and 21 features for those in AMD dataset, as listed in Table 4.4.

Table 4.4: Selected features for the Drebin and AMD datasets.

FEATURE	DREBIN	AMD
UNIQUE_IDENTIFIER → SMS_MMS	✓	✓
UNIQUE_IDENTIFIER → ACCOUNT_SETTINGS	✓	✓
UNIQUE_IDENTIFIER → AUDIO	✓	✓
LOCATION_INFORMATION → LOCATION_INFORMATION	✓	✓
ACCOUNT_INFORMATION → CONTACT_INFORMATION	✓	✓
ACCOUNT_INFORMATION → AUDIO	✓	✓
ACCOUNT_INFORMATION → EMAIL	✓	✓
BROWSER_INFORMATION → BROWSER_INFORMATION	✓	✓
NETWORK_INFORMATION → SMS_MMS	✓	✓
NETWORK_INFORMATION → BLUETOOTH	✓	✓
SYSTEM_SETTINGS → LOG	✓	
SYSTEM_SETTINGS → LOCATION_INFORMATION	✓	✓
SYSTEM_SETTINGS → BLUETOOTH	✓	✓
SYSTEM_SETTINGS → BROWSER_INFORMATION	✓	✓
FILE_INFORMATION → SMS_MMS	✓	✓
CONTACT_INFORMATION → LOG	✓	✓
DATABASE_INFORMATION → FILE	✓	
SYNCHRONIZATION_DATA → FILE	✓	✓
EMAIL → FILE	✓	✓
EMAIL → AUDIO	✓	✓
EMAIL → NO_CATEGORY		✓
CALENDAR_INFORMATION → LOG	✓	✓
SMS_MMS → NO_CATEGORY	✓	
NO_CATEGORY → AUDIO	✓	✓



(a) Drebin dataset



(b) AMD dataset

Figure 4.6: Feature selection trials for different values of threshold.

4.3.3 Clustering

We considered k-means as clustering algorithm using the Euclidean distance, as it is one of the oldest and most widely used unsupervised learning algorithms [190]. k-means is a heuristic algorithm which partitions a dataset into a number of clusters by minimizing the sum of squared distances between each pair of the clusters. Despite its benefits, it has two potential shortcomings: it has a super-polynomial execution time in the worst case, and the solution found can be arbitrarily bad compared to an optimal clustering [191]. To overcome these issues, we leveraged an optimal vari-

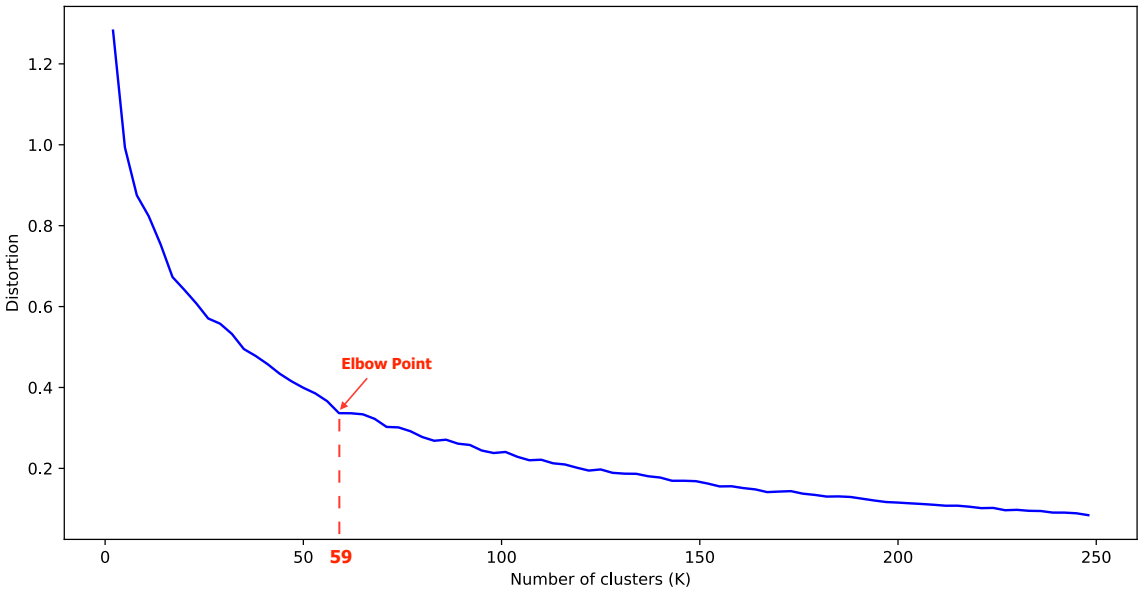
ant of k-means, known as k-means++ [192], in which initial values (seeds) are set so the algorithm finds an optimal solution in logarithmic time.

One key aspect when using k-means++ is determining the target amount of clusters, which can be done by using a clustering evaluation method. The basic idea behind clustering evaluation methods is to choose the number of clusters that optimize a specific criterion. The so-called elbow method is one of the most popular techniques and can be used with various clustering quality criteria, including distortion, Silhouette [193] and Calinski-Harabasz [194]. We used the elbow method using the distortion, which tries to find the optimal number of clusters by minimizing the average sum of squared distances to each cluster's centroid. Our choice is motivated by its speed, as it proved to be much faster than all others and this turned out to be critical during our experimentation. Figures 4.7a and 4.7b show the results of this method over the two datasets, which yielded 59 and 22 clusters for Drebin and AMD, respectively.

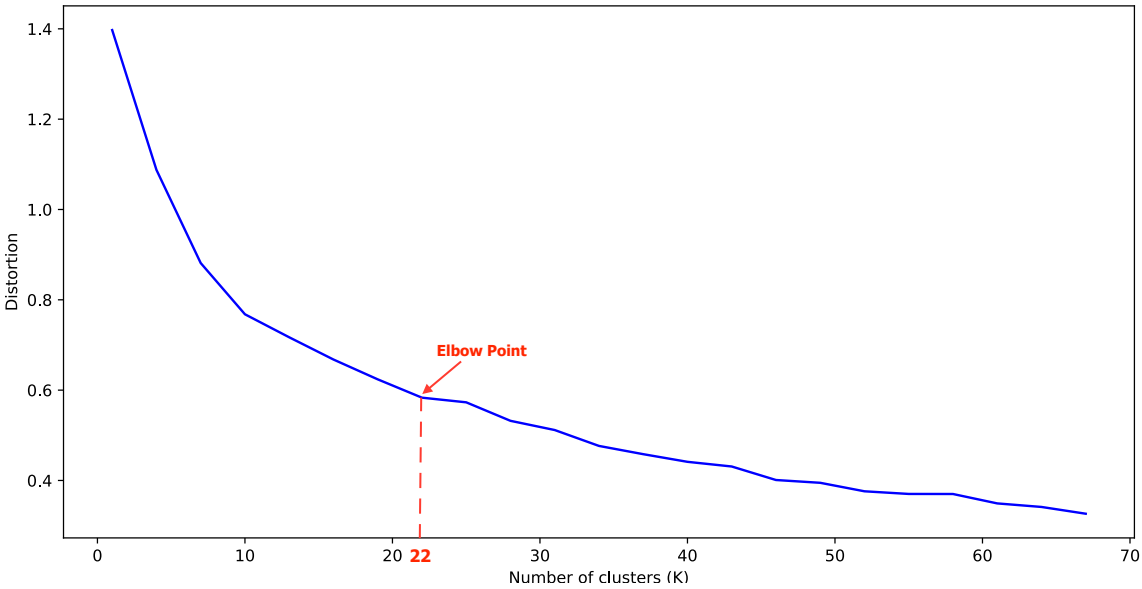
4.3.4 Behavioral Relevance of Clusters and Relation to Families

After clustering the samples in each dataset using the optimal number of clusters determined above, we addressed two separate questions: are these groups meaningful from a behavioral point of view? Are they linked to current family labels?

Figs 4.8 and 4.9 show the centroids of all clusters for Drebin and AMD datasets considering their features at stake (recall Section 4.3.1). As it can be seen, each cluster holds a different combination of prevalence among all features, thus confirming that clusters are behaviorally different. It is notable that some features are present in most clusters. For example, *LOCATION_INFORMATION* $\rightarrow$ *LOCATION_INFORMATION* is prominent in 43 clusters of Drebin and 15 clusters of AMD. Also, *ACCOUNT_INFORMATION* $\rightarrow$ *Email* is heavily present in 25 and 9 clusters of Drebin and AMD datasets respectively. Here, *AUDIO*, *SMS_MMS* and *FILE* appear as the most common sink categories among all clusters in both datasets. On the other hand, there are features which are specific to a limited number of clusters. For example, *CALENDAR_INFORMATION* $\rightarrow$ *LOG* is highly frequent in cluster 42 and also present in cluster 23 of Drebin exclusively. As another example, *NETWORK_INFORMATION* $\rightarrow$



(a) Drebin dataset



(b) AMD dataset

Figure 4.7: Elbow evaluation to select the optimal number of clusters.

BLUETOOTH is highly present in cluster 11 and is frequent in cluster 20 of AMD specifically.

To study the connection between clusters and family labels, we analyzed the distribution of original families into our behavioral-based clusters (Figs. 4.10a and 4.10b). As it can be seen, only a fraction of families (e.g., FakeLogo in Drebin and Boxer in AMD) have all their samples in a single cluster. On the contrary, in most cases families are spread across

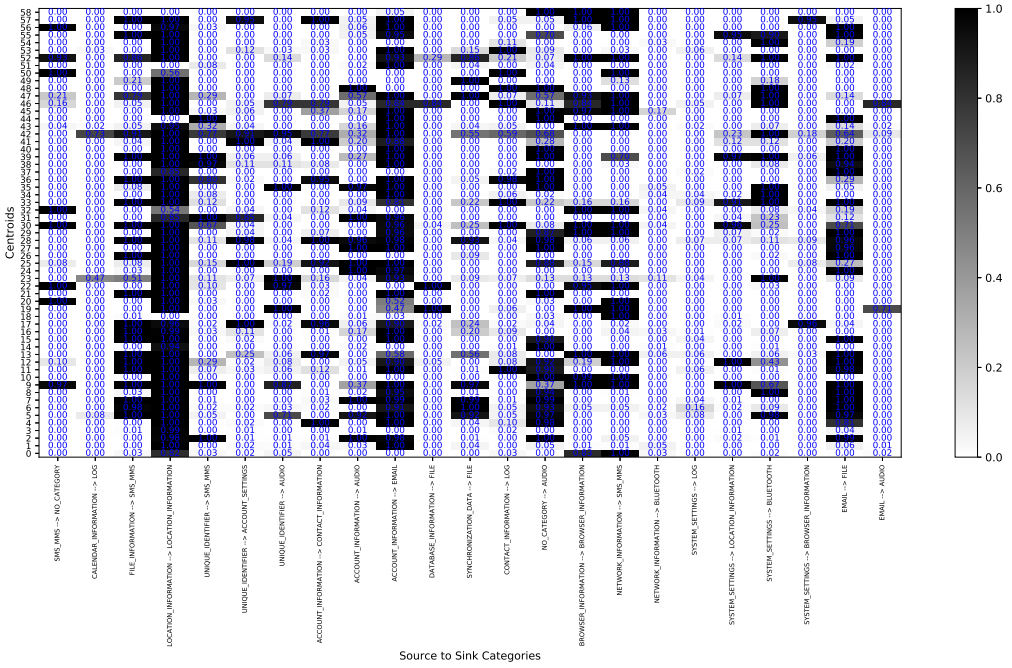


Figure 4.8: Centroids obtained after clustering the Drebin dataset.

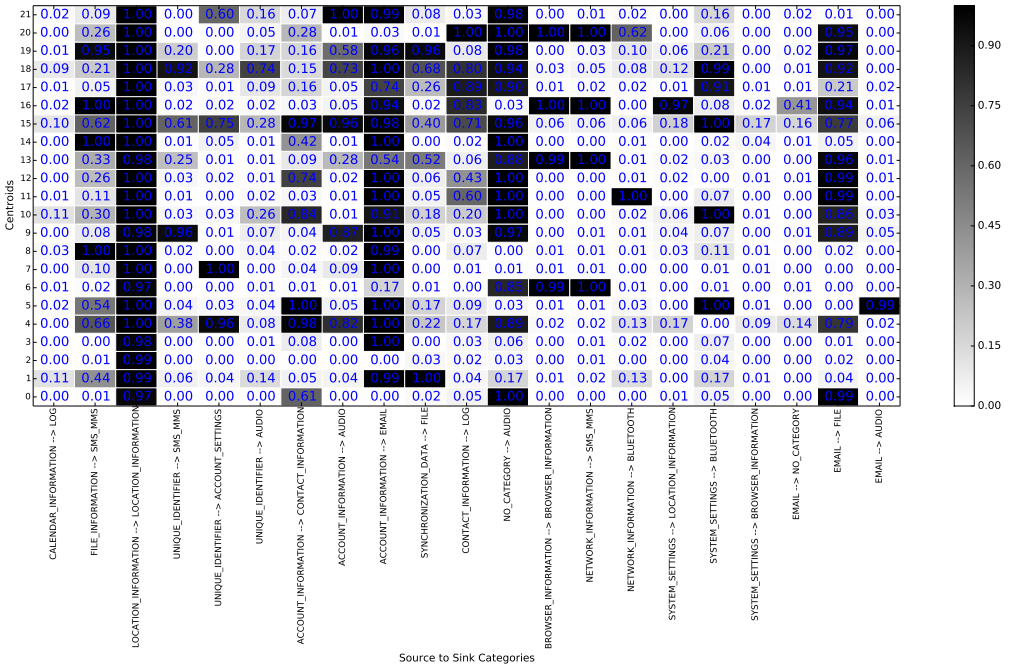
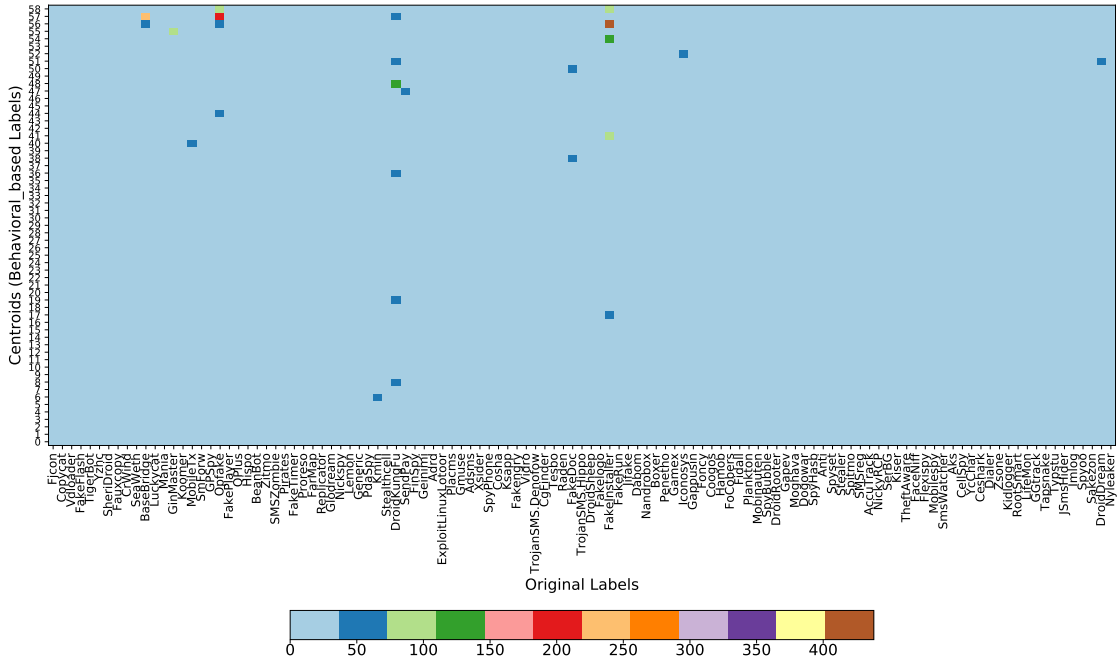


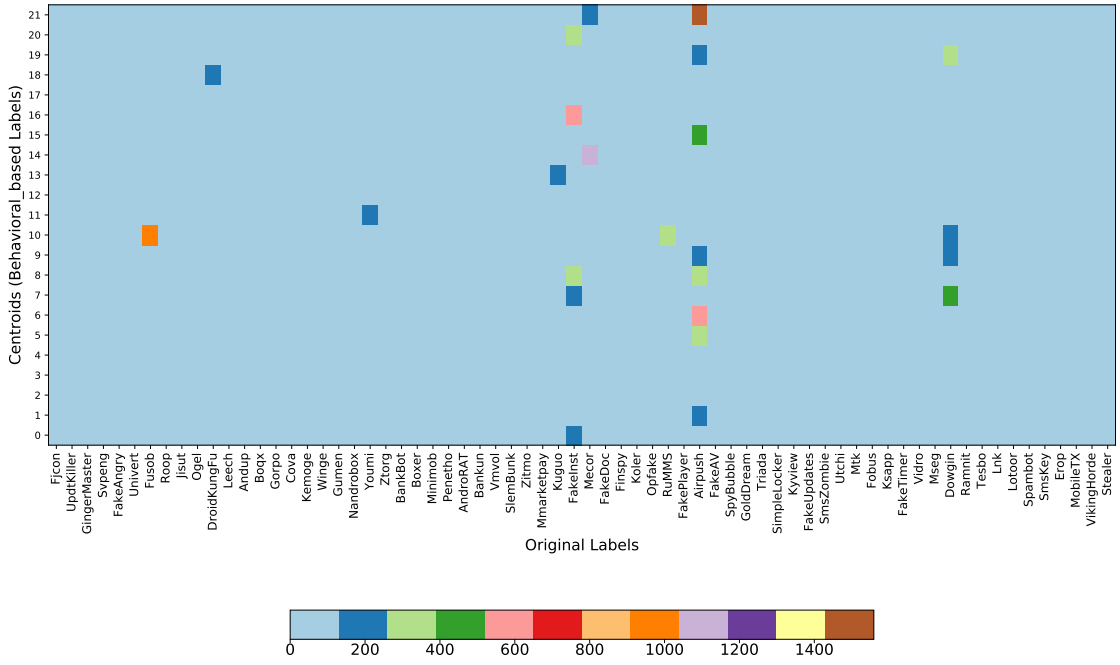
Figure 4.9: Centroids obtained after clustering the AMD dataset.

clusters, and, conversely, most clusters contain samples of many families.

From another perspective, we have also measured (Figs. 4.11 and 4.12) the number of clusters in which each family is present. The higher the amount, the more behavioral profiles exist in that family. In this regard,



(a) Drebin dataset



(b) AMD dataset

Figure 4.10: Distribution of samples into clusters.

smaller families had commonly one behavioral profile, and, therefore, were classified into a single cluster. On the contrary, we observed more behavioral differences in large families. Thus, DroidKungFu, GinMaster, Base-

Bridge and Plankton families from Drebin dataset are scattered across 34, 26, 22 and 16 clusters, respectively. Similarly, the apps from Airpush, Dowgin, Youmi and Kuguo families of AMD dataset are present in 22, 19, 16 and 15 different clusters. However, there are some exceptions to this general pattern. For instance, the Iconosys and FakeDoc families of Drebin, with 151 and 132 apps, respectively, have fewer behavioral differences than families such as Glodream (44 apps) or Plankton (59 apps). Furthermore, BankBot and SlemBunk of AMD dataset, two well-known Trojan-Banker families, each of them with 323 and 127 applications, had only 5 behavioral profiles. SimpleLocker and Fusob, two popular ransomware, each of which with 98 and 965 apps had only 3 and 2 various behavioral profiles respectively. Finally, a Trojan-Spy family, known as Mecor, with 1,383 apps had only 3 types of behavioral profiles. All these findings support the claim that current family labels might be unrelated to the real behavior of their samples.

4.3.5 Examples

We next discuss examples belonging to the four possible cases depending on apps behavior (i.e., same/different cluster) and their classification (i.e. same/different family). Naturally, the case in which apps behave differently and belong to different families would not lead us to any conclusion about the behavioral relevance of families, and, therefore, we focus on the remaining cases.

4.3.5.1 Same Family, Same Behavior

There are applications that belong to the same family and are classified in the same cluster. This reduced subset represents those cases in which the concept of family is related to the actual behavior. Table 4.5 shows three examples of this case from Drebin dataset along with an some of their information flows. The two apps from *FakeInstaller* family are Trojans which request four sensitive permission upon installation and try to leak smartphone's sensitive information through the network. The apps from *Yzhc* family are again Trojans which try to call API methods that provide access to information about the telephony services on the device. Applications can use these methods to determine telephony services and states, as well as to access some types of subscriber information. Finally, malware

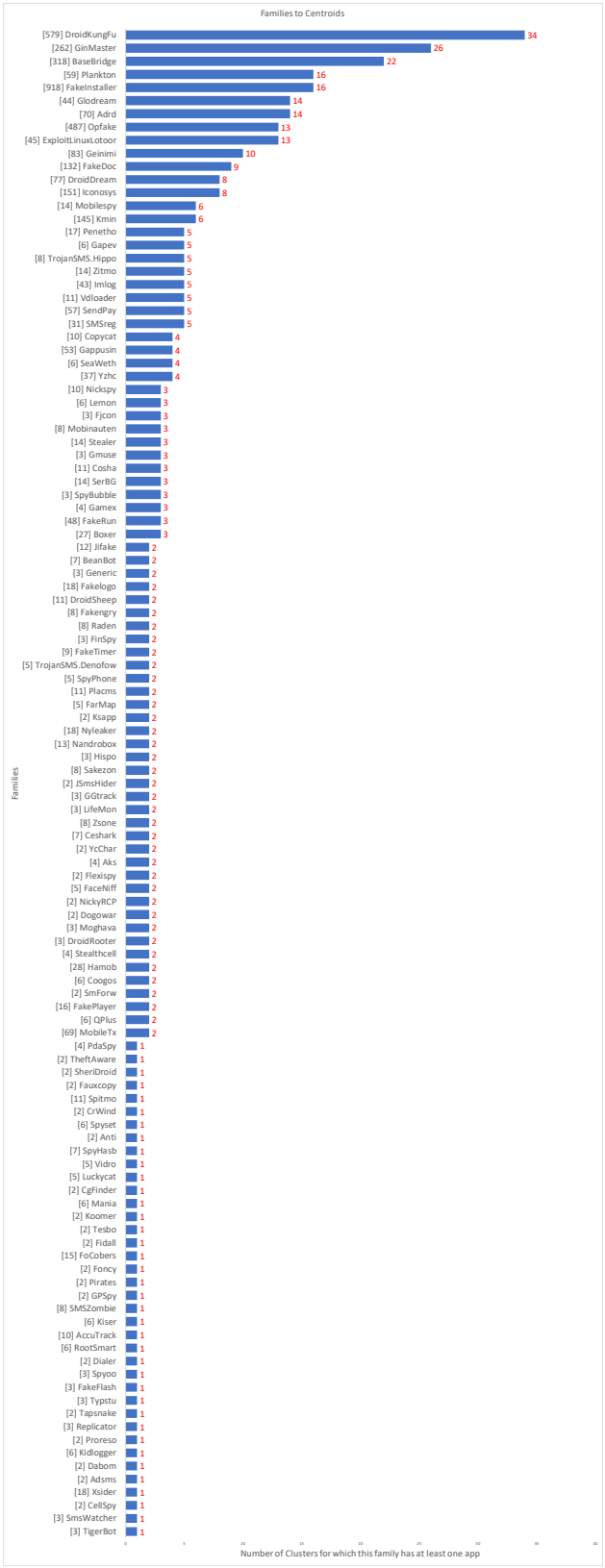


Figure 4.11: Number of clusters in which each family of the Drebin dataset is present.

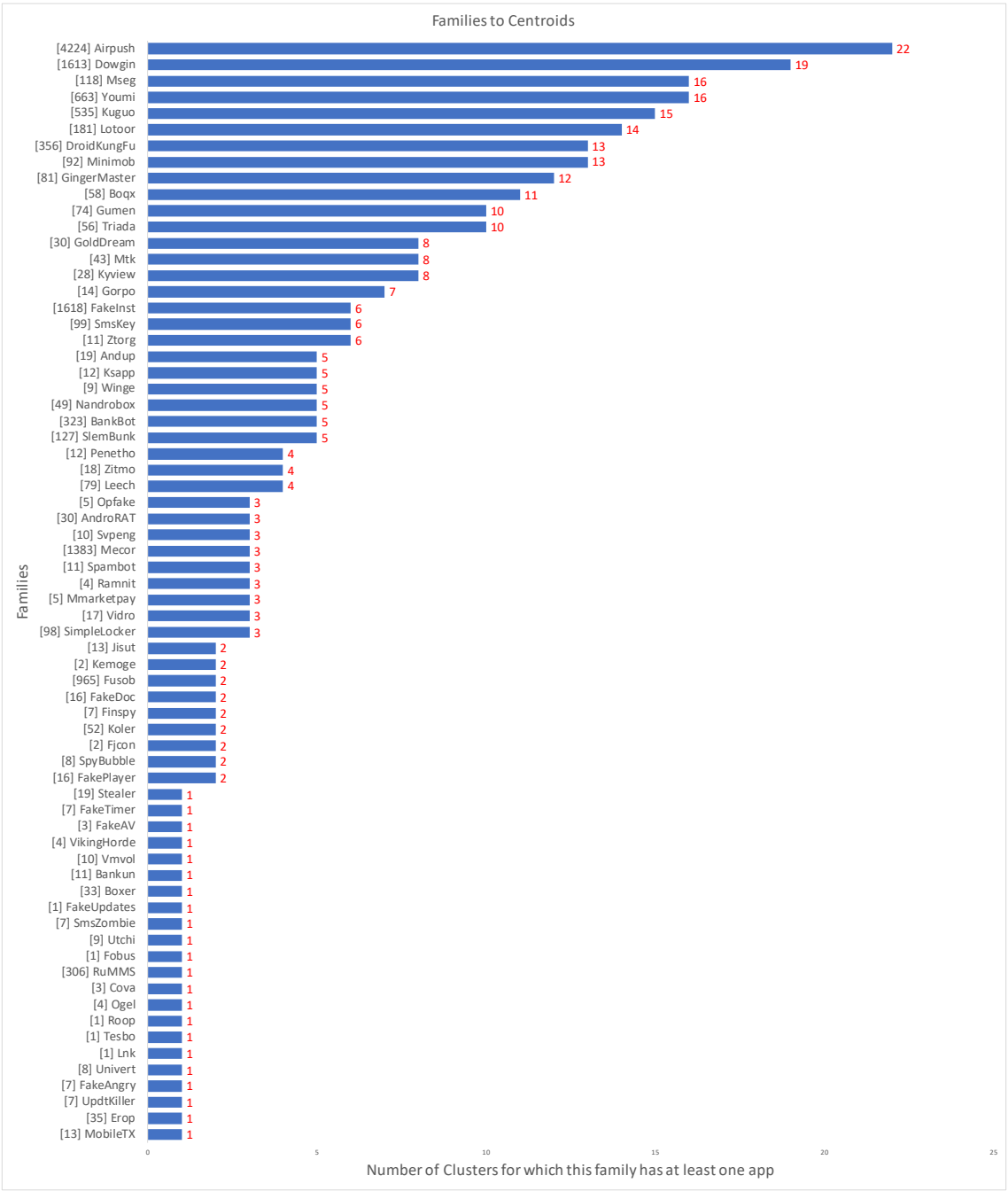


Figure 4.12: Number of clusters in which each family of the AMD dataset is present.

samples from *ExploitLinuxLotoor* family are repackaged apps which upon installation try to root and exploit the device. Similar examples are also found in the AMD dataset.

Table 4.5: Examples of samples from the same family of Drebin dataset exhibiting similar behaviors.

Family Name	App 1	Flows in App 1	App 2	Flows in App 2
FakeInstaller	dbc4b035...4e8e.apk	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getLine1Number() → Log.v()* NETWORK_INFORMATION → LOG: *TelephonyManager.getSimOperator() → Log.v()* *TelephonyManager.getSimCountryIso() → Log.i()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getLine1Number() → Log.v()* NETWORK_INFORMATION → NO_CATEGORY: *TelephonyManager.getSimCountryIso() → String.substring()* NO_CATEGORY → LOG: *HashMap.get() → Log.d()* *HashMap.get() → Log.v()* *HashMap.get() → Log.i()* NO_CATEGORY → NO_CATEGORY: *HashMap.get() → String.substring()* *HashMap.get() → HashMap.put()*	0281dcaf...4683b.apk	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getLine1Number() → Log.v()* NETWORK_INFORMATION → LOG: *TelephonyManager.getSimOperator() → Log.v()* *TelephonyManager.getSimCountryIso() → Log.i()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getLine1Number() → Log.v()* NETWORK_INFORMATION → NO_CATEGORY: *TelephonyManager.getSimCountryIso() → String.substring()* NO_CATEGORY → LOG: *HashMap.get() → Log.d()* *HashMap.get() → Log.v()* *HashMap.get() → Log.i()* NO_CATEGORY → NO_CATEGORY: *HashMap.get() → String.substring()* *HashMap.get() → HashMap.put()*
Yzhc	c7a30af7...3938.apk	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getDeviceId() → Log.v()* *TelephonyManager.getLine1Number() → Log.v()* *TelephonyManager.getSimSerialNumber() → Log.v()* NO_CATEGORY → LOG: *Intent.getAction() → Log.v()*	7a21caba...b021.apk	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getDeviceId() → Log.v()* *TelephonyManager.getLine1Number() → Log.v()* *TelephonyManager.getSimSerialNumber() → Log.v()* NO_CATEGORY → LOG: *Intent.getAction() → Log.v()*
ExploitLinuxLotoor	bee54fae...5e70.apk	NO_CATEGORY → LOG: *Runtime.getRuntime() → Log.d()* *File.getAbsolutePath() → Log.e()* *File.getAbsolutePath() → Log.d()* *EncodingUtils.getAsciiString() → Log.d()* NO_CATEGORY → FILE: *Resources.getAssets() → FileOutputStream.write()* NO_CATEGORY → NO_CATEGORY: *Resources.getAssets() → AssetManager.open()* *File.getAbsolutePath() → MediaPlayer.setDataSource()* *Resources.getString() → Intent.putExtra()*	4c70f5ef...d27e.apk	NO_CATEGORY → LOG: *Runtime.getRuntime() → Log.d()* *File.getAbsolutePath() → Log.e()* *File.getAbsolutePath() → Log.d()* *EncodingUtils.getAsciiString() → Log.d()* NO_CATEGORY → FILE: *Resources.getAssets() → FileOutputStream.write()* NO_CATEGORY → NO_CATEGORY: *Resources.getAssets() → AssetManager.open()* *File.getAbsolutePath() → MediaPlayer.setDataSource()* *Resources.getString() → Intent.putExtra()*

4.3.5.2 Different Family, Same Behavior

The second case involves applications that behave similarly (i.e., belong to the same cluster) but are classified into different families. Table 4.7 includes some examples from the Drebin dataset. The first pair of apps are from *Opfake* and *Stealer* families. Here, the app from *Opfake* family is a Trojan which infects users when they click on a link that allows opening a Java enabled browser or web page. Then, the device performs the malicious activities indicated in a given web page. These commands enable attackers to install malicious software, steal personal information, or send text messages depending on the variant type. The app from the *Stealer*

Table 4.6: Examples of samples from the same family of AMD dataset exhibiting similar behaviors.

Family Name	App 1	Flows in App 1	App 2	Flows in App 2
SimpleLocker	401935ef...1753.apk	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* *SmsManager.divideMessage() → SmsManager.sendMultipartTextMessage()* *SmsManager.getDefault() → SmsManager.sendMultipartTextMessage()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getDeviceId() → JSONObject.put()* NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getMessageBody() → JSONObject.put()* *SmsMessage.getOriginatingAddress() → JSONObject.put()* NO_CATEGORY → SMS_MMS: *JSONObject.get() → SmsManager.sendTextMessage()* *JSONObject.getString() → SmsManager.sendTextMessage()* *JSONObject.get() → SmsManager.sendMultipartTextMessage()* *JSONObject.getString() → SmsManager.sendMultipartTextMessage()* NO_CATEGORY → FILE: *File.getAbsolutePath() → FileOutputStream.write()* NO_CATEGORY → NO_CATEGORY: *Camera.getParameters() → Camera.setParameters()* *JSONArray.getString() → HashSet.add()* *JSONObject.get() → HashSet.add()* *JSONObject.getString() → HashSet.add()* *PowerManager.newWakeLock() → JSONObject.put()* *JSONObject.get() → JSONObject.put()* *Camera$Parameters.getSupportedPreviewSizes() → Camera$Parameters.setPictureSize()* *Camera.getParameters() → Camera.setPictureSize()* *JSONObject.get() → String.substring()* *JSONArray.getString() → ObjectOutputStream.writeObject()* *JSONObject.get() → ObjectOutputStream.writeObject()* *JSONObject.getString() → ObjectOutputStream.writeObject()* *JSONObject.getString() → JSONObject.put()* *File.getAbsolutePath() → String.substring()*	cfa03953...a5a7.apk	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* *SmsManager.divideMessage() → SmsManager.sendMultipartTextMessage()* *SmsManager.getDefault() → SmsManager.sendMultipartTextMessage()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getDeviceId() → JSONObject.put()* NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getMessageBody() → JSONObject.put()* *SmsMessage.getOriginatingAddress() → JSONObject.put()* NO_CATEGORY → SMS_MMS: *JSONObject.get() → SmsManager.sendTextMessage()* *JSONObject.getString() → SmsManager.sendTextMessage()* *JSONObject.get() → SmsManager.sendMultipartTextMessage()* *JSONObject.getString() → SmsManager.sendMultipartTextMessage()* NO_CATEGORY → FILE: *File.getAbsolutePath() → FileOutputStream.write()* NO_CATEGORY → NO_CATEGORY: *Camera.getParameters() → Camera.setParameters()* *JSONArray.getString() → HashSet.add()* *JSONObject.get() → HashSet.add()* *JSONObject.getString() → HashSet.add()* *PowerManager.newWakeLock() → JSONObject.put()* *JSONObject.get() → JSONObject.put()* *Camera$Parameters.getSupportedPreviewSizes() → Camera$Parameters.setPictureSize()* *Camera.getParameters() → Camera.setPictureSize()* *JSONObject.get() → String.substring()* *JSONArray.getString() → ObjectOutputStream.writeObject()* *JSONObject.get() → ObjectOutputStream.writeObject()* *JSONObject.getString() → ObjectOutputStream.writeObject()* *JSONObject.getString() → JSONObject.put()* *File.getAbsolutePath() → String.substring()*
SlemBunk	b4a1b355...9b0a.apk	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* *SmsManager.getDefault() → SmsManager.sendMultipartTextMessage()* *SmsManager.divideMessage() → SmsManager.sendMultipartTextMessage()* NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getMessageBody() → JSONObject.put()* *SmsMessage.getOriginatingAddress() → JSONObject.put()* *SmsMessage.getMessageBody() → String.substring()* *SmsMessage.getOriginatingAddress() → String.substring()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getDeviceId() → JSONObject.put()* CALENDAR_INFORMATION → NO_CATEGORY: *Calendar.getInstance() → Calendar.setTime()* *Calendar.getTimeInMillis() → AlarmManager.set()* NO_CATEGORY → AUDIO: *AnimationUtils.loadAnimation() → AudioManager.setRingerMode()* *ViewGroup.getChildAt() → AudioManager.setRingerMode()* *EditText.getText() → AudioManager.setRingerMode()* NO_CATEGORY → SMS_MMS: *ViewGroup.getChildAt() → SmsManager.sendTextMessage()* *EditText.getText() → SmsManager.sendTextMessage()* *JSONObject.getString() → SmsManager.sendTextMessage()* *AnimationUtils.loadAnimation() → SmsManager.sendTextMessage()* *JSONObject.get() → SmsManager.sendTextMessage()* *AnimationUtils.loadAnimation() → SmsManager.sendMultipartTextMessage()* *ViewGroup.getChildAt() → SmsManager.sendMultipartTextMessage()* *EditText.getText() → SmsManager.sendMultipartTextMessage()* NO_CATEGORY → NO_CATEGORY: *Throwable.getMessage() → JSONObject.put()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *Locale.getCountry() → JSONObject.put()* *ContentResolver.query() → JSONObject.put()* *TextView.getText() → AlarmManager.set()*	b77fae69...1405.apk	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* *SmsManager.getDefault() → SmsManager.sendMultipartTextMessage()* *SmsManager.divideMessage() → SmsManager.sendMultipartTextMessage()* NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getMessageBody() → JSONObject.put()* *SmsMessage.getOriginatingAddress() → JSONObject.put()* *SmsMessage.getMessageBody() → String.substring()* *SmsMessage.getOriginatingAddress() → String.substring()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getDeviceId() → JSONObject.put()* CALENDAR_INFORMATION → NO_CATEGORY: *Calendar.getInstance() → Calendar.setTime()* *Calendar.getTimeInMillis() → AlarmManager.set()* NO_CATEGORY → AUDIO: *AnimationUtils.loadAnimation() → AudioManager.setRingerMode()* *ViewGroup.getChildAt() → AudioManager.setRingerMode()* *EditText.getText() → AudioManager.setRingerMode()* NO_CATEGORY → SMS_MMS: *ViewGroup.getChildAt() → SmsManager.sendTextMessage()* *EditText.getText() → SmsManager.sendTextMessage()* *JSONObject.getString() → SmsManager.sendTextMessage()* *AnimationUtils.loadAnimation() → SmsManager.sendTextMessage()* *JSONObject.get() → SmsManager.sendTextMessage()* *AnimationUtils.loadAnimation() → SmsManager.sendMultipartTextMessage()* *ViewGroup.getChildAt() → SmsManager.sendMultipartTextMessage()* *EditText.getText() → SmsManager.sendMultipartTextMessage()* NO_CATEGORY → NO_CATEGORY: *Throwable.getMessage() → JSONObject.put()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *Locale.getCountry() → JSONObject.put()* *ContentResolver.query() → JSONObject.put()* *TextView.getText() → AlarmManager.set()*

family is also a kind of Trojan leaking personal information through the network. Also, the pair of apps from *RuFraud* and *Zitmo* families share several information flows. Thus, the sample from *RuFraud* is a Trojan which send premium-rate SMS messages to a number depending on the country where the SIM card is registered. Also, it hides any messages re-

ceived from the target numbers. On the other hand, the sample from *Zitmo* is a banking Trojan which tries to defeat the second factor of authentication in an electronic transaction and steal credentials and financial data. Finally, the apps from *MobileTx* and *YcChar* are also Trojans which steal personally identifiable information from the device, log them and sometimes send them to a server or premium-rate number. Table 4.8 contains two instances for the same case from AMD dataset. The first pair of apps are from Gumen and Leech families detected in 2013 and 2015, respectively. Here, the app from Gumen family is a Trojan which mainly logs keyboard strokes and collects system information in order to leak them to a remote server. It also has the ability to run or terminate other processes after obtaining root privileges. Similarly, the app from Leech family is also a Trojan which tries to get root privileges initially and inject itself into other system processes by exploiting the Zygote process, the core process in the Android OS. After gaining root privileges, it tries to collect a variety of information and sends them to a remote server. This family along with two other families, Ztorg and Gorpo, form the basic modules of a sophisticated and recently detected Android Trojan family known as Triada. The second pair of apps are from Erop and FakeIns families, both known to be Trojan-SMS malware. The app from Erop family penetrates the devices via users downloads. It then starts running in the background when the browser is opened. Next, it starts capturing information and advertising unwanted services by sending premium rate text messages. Also, the app from FakeIns family is a Trojan able to send premium text messages in more than 60 countries. This app disguises itself as a legitimate Android application to watch pornographic movies. It can also delete, respond or even intercept incoming text messages.

4.3.5.3 Same Family, Different Behavior

The third case happens when two applications of the same family behave different. Table 4.9 lists three examples of this case from the Drebin dataset. The first pair of apps are from a well-known Trojan family, called *BaseBridge*. Generally, this family is known to have malware that once installed can achieve root privileges and install their payloads. Once the device is infected, they can send sensitive unique identifier information to remote control servers or to premium numbers. However, *App 1* in Table 4.9 contains source and sink API methods which are not sensitive whereas

Table 4.7: Examples of samples in different families of Drebin dataset with similar behaviors.

App 1	Family of App 1	Flows in App 1	App 2	Family of App 2	Flows in App 2
f24beb...9bc.apk	Opfake	NETWORK_INFORMATION → NO_CATEGORY: *TelephonyManager.getNetworkOperator() →String.startsWith()* NO_CATEGORY → NO_CATEGORY: *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *Hashtable.get() → PrintStream.println()* *PendingIntent.getBroadcast() → AlarmManager.setRepeating()*	6a8069...0ce.apk	Stealer	NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getOriginatingAddress() →Intent.putExtra()* *SmsMessage.getMessageBody() → Intent.putExtra()* *SmsMessage.getMessageBody() → String.startsWith()* NO_CATEGORY → NO_CATEGORY: *Intent.getExtras() → Intent.putExtra()* *PendingIntent.getBroadcast() → AlarmManager.set()*
477d6b...892.apk	RuFraud	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* NETWORK_INFORMATION → NO_CATEGORY: *TelephonyManager.getNetworkOperator() → String.substring()* NO_CATEGORY → SMS_MMS: *PendingIntent.getBroadcast() → SmsManager.sendTextMessage()* *JSONObject.getString() → SmsManager.sendTextMessage()*	1f4092...240.apk	Zitmo	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* *SmsMessage.getMessageBody() → SmsManager.sendTextMessage()* *SmsMessage.getOriginatingAddress() → SmsManager.sendTextMessage()* NETWORK_INFORMATION → NO_CATEGORY: *SmsMessage.getMessageBody() → String.substring()* *SmsMessage.getMessageBody() → String.startsWith()* NO_CATEGORY → SMS_MMS: *PendingIntent.getBroadcast() → SmsManager.sendTextMessage()*
efec59...523.apk	MobileTx	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getSubscriberId() → Log.i()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getSubscriberId() → Intent.putExtra()* NO_CATEGORY → LOG: *ByteArrayOutputStream.toByteArray() → Log.i()* *Intent.getStringExtra() → Log.i()* NO_CATEGORY → NO_CATEGORY: *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *Bitmap.createBitmap() → ImageView.setImageBitmap()* *JSONObject.getString() → AssetManager.open()* *String.getBytes() → OutputStream.write()*	6b7e68...4e4.apk	YcChar	UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getDeviceId() → Log.i()* DATABASE_INFORMATION → LOG: *SQLiteDatabase.query() → Log.d()* *SQLiteDatabase.getPath() → Log.d()* NO_CATEGORY → LOG: *Properties.getProperty() → Log.i()* NO_CATEGORY → NO_CATEGORY: *Properties.getProperty() → AssetManager.open()*

App 2 is extracting identifier information such as *getSimSerialNumber*. Another case happens in *Glodream* where *App 1* logs the SIM (Subscriber Identity Module) serial number comparing with *App 2* that leaks sensitive information through sending text messages. The third pair of apps come from the popular Trojan family, *Nickspy*, that leaks sensitive information by sending text messages. However, *App 1* leaks location information in this way, while most flows in *App 2* are non-sensitive. Table 4.10 shows two examples of the same case from AMD dataset. The first pair of apps are from Triada family which was first detected in 2016. Triada is one of the most advanced types of Trojan which is downloaded and installed by three other previously known Trojan families, including Leech, Gorpo and Ztorg. After obtaining root privileges, this Trojan can leverage the Zygote process and can be pre-installed into any applications running on the smartphone and change their logics. It is a modular Trojan with a wide range of capabilities which are all set by a single command through C&C servers.

Table 4.8: Examples of samples in different families of AMD dataset with similar behaviors.

App 1	Family of App 1	Flows in App 1	App 2	Family of App 2	Flows in App 2
0f507d...511.apk	Leech	NO_CATEGORY → LOG: *Log.getStackTraceString() → Log.println()* *Class.getSimpleName() → Log.println()* *Throwable.getMessage() → Log.println()* NO_CATEGORY → NETWORK: *Bundle.getString() → URL.openConnection()* NO_CATEGORY → NO_CATEGORY: *Bundle.getString() → Intent.putExtra()* *Charset.name() → String.startsWith()* *Resources.getColor() → View.setBackgroundColor()* *Charset.name() → String.substring()* *Intent.getIntExtra() → Activity.onCreate()* *Resources.getStringArray() → Activity.onCreate()* *LruCache.get() → ImageView.setImageBitmap()*	1d0f43...488.apk	Gumen	NO_CATEGORY → LOG: *JSONObject.getString() → Log.v()* *Locale.getISO3Country() → Log.v()* *Class.getSimpleName() → Log.d()* *Class.getName() → Log.d()* *TextView.getText() → Log.e()* *Settings$Secure.getString() → Log.v()* NO_CATEGORY → NETWORK: *JSONObject.getString() → URL.openConnection()* NO_CATEGORY → NO_CATEGORY: *URLConnection.getInputStream() → OutputStream.write()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *LinkedHashMap.get() → Writer.write()* *PendingIntent.getBroadcast() → AlarmManager.set()*
9a8259...87b.apk	Erop	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* NO_CATEGORY → SMS_MMS: *JSONObject.getString() → SmsManager.sendTextMessage()* *JSONObject.get() → SmsManager.sendTextMessage()* NO_CATEGORY → NO_CATEGORY: *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* *WebView.getSettings() → WebSettings.setUseWideViewPort()* *WebView.getSettings() → WebSettings.setLoadWithOverviewMode()* *WebView.getSettings() → WebSettings.setBuiltInZoomControls()*	0038be...a70.apk	FakeInst	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* NO_CATEGORY → SMS_MMS: *PendingIntent.getBroadcast() → SmsManager.sendTextMessage()* *ArrayList.get() → SmsManager.sendTextMessage()* NO_CATEGORY → NO_CATEGORY: *Resources.getXml() → HashMap.put()* *PendingIntent.getBroadcast() → AlarmManager.set()*

However, App 1 is only capturing sensitive unique identifier information and saving them into JSON files to leak them later in various ways whereas App 2 does not contain any sensitive information flows. The second pair of apps are from Fusob family, a ransomware first detected in 2015. This family is still the most active ransomware in Germany [183] while it infects very few users in Russia, Ukraine and Kazakhstan based on the language which is active on the device. It displays fake screens and tries to accuse victims for crimes. Then, it threatens them that a criminal case will be opened unless they pay the fine. However, from the two apps listed in Table 4.10, App 1 has much more capabilities (or more sensitive information flows) than App 2. Here, App 1 extracts four main critical information related to the network and unique identifiers, including Service Provider Name or SPN (*TelephonyManager.getSimOperatorName()*), phone number string for line 1 (*TelephonyManager.getLine1Number()*), device ID (*TelephonyManager.getDeviceId()*) and subscriber ID such as the International Mobile Subscriber Identity (IMSI) for a GSM phone (*TelephonyManager.getSubscriberId()*). It then saves all of them into JSON files

Table 4.9: Examples of samples in the same family of Drebin dataset exhibiting different behaviors.

Family Name	App 1	Flows in App 1	App 2	Flows in App 2
BaseBridge	0b57c267...404f.apk	NO_CATEGORY → NO_CATEGORY: *Resources.getIdentifier() → HashMap.put()* *Resources.getIdentifier() → AlertDialog$Builder.setSingleChoiceItems()* *Resources.getIdentifier() → AlertDialog$Builder.setTitle()* *Resources.getIdentifier() → AlertDialog$Builder.setNegativeButton()* *Resources.getIdentifier() → AlertDialog$Builder.setMessage()* *Resources.getString() → Intent.putExtra()* *Resources.getStringArray() → Intent.putExtra()* *Context.getString() → String.substring()* *Context.getString() → HashMap.put()* *Resources.getIdentifier() → AlertDialog$Builder.setPositiveButton()* *Bitmap.createBitmap() → ImageView.setImageBitmap()*	28aac623...e4eb.apk	UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getSimSerialNumber() → String.startsWith()* *TelephonyManager.getSimSerialNumber() → String.substring()*
Glodream	c2b2c2be...21eb.apk	UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getSimSerialNumber() → String.startsWith()* *TelephonyManager.getSimSerialNumber() → String.substring()* NO_CATEGORY → LOG: *Bundle.getString() → Log.d()* *Bundle.getString() → Log.e()* *Bundle.getInt() → Log.d()* NO_CATEGORY → NO_CATEGORY: *Resources.getDrawable() → Drawable.setBounds()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()*	91ee43dd...3bc1.apk	NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()* NO_CATEGORY → LOG: *EditText.getText() → Log.w()* *Array.newInstance() → Log.w()* NO_CATEGORY → SMS_MMS: *String.getBytes() → SmsManager.sendTextMessage()* NO_CATEGORY → NETWORK: *String.getBytes() → URLConnection.openConnection()* NO_CATEGORY → NO_CATEGORY: *String.getBytes() → String.substring()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()*
Nicksspy	498b425a...5008.apk	LOCATION_INFORMATION → SMS_MMS: *Location.getLatitude() → SmsManager.sendTextMessage()* *Location.getLongitude() → SmsManager.sendTextMessage()* *GsmCellLocation.getCid() → SmsManager.sendTextMessage()* *GsmCellLocation.getLac() → SmsManager.sendTextMessage()* NETWORK_INFORMATION → SMS_MMS: *SmsManager.getDefault() → SmsManager.sendTextMessage()*	cdec039a...af03.apk	NO_CATEGORY → LOG: *InetAddress.getByName() → Log.d()* NO_CATEGORY → NO_CATEGORY: *PendingIntent.getBroadcast() → AlarmManager.setRepeating()* *EditText.getText() → String.substring()*

(*JSONObject.put()*) for later exfiltration. On the other side, App 2 does not contain any of these sensitive flows.

4.3.5.4 Summary

All in all, our analysis reveals that existing families do not exhibit behavioral consistency. We spotted samples in the same family with very different behavior, as well as samples in different families with behavioral similarity. Moreover, our clustering procedure has shown that it is possible to divide datasets into groups with different behavioral footprints. Finally, to show the time and memory complexity of our characterization method, we have measured the average amount of time and memory (Table 4.11) which is consumed in each step of our clustering approach per application. As it is clear, extracting information flows, and, later, flows patterns are

Table 4.10: Examples of samples in the same family of AMD dataset exhibiting different behaviors.

Family Name	App 1	Flows in App 1	App 2	Flows in App 2
Triada	9cbe79ce...5b65.apk	NO_CATEGORY → NO_CATEGORY: *JSONArray.getString() → String.startsWith()* *ActivityManager.getRunningAppProcesses() → String.startsWith()* *JSONObject.getInt() → JSONObject.put()* *JSONObject.getString() → JSONObject.put()* *Throwable.getStackTrace() → JSONObject.put()* *Throwable.getMessage() → JSONObject.put()* *ArrayList.get() → String.substring()* *ByteArrayOutputStream.toByteArray() → OutputStream.write()* *Properties.getProperty() → JSONObject.put()* *ArrayList.get() → JSONObject.put()* *ComponentName.getPackageName() → HashMap.put()* *String.getBytes() → RandomAccessFile.write()* UNIQUE_IDENTIFIER → LOG: *TelephonyManager.getSimSerialNumber() → Log.i()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getDeviceId() → JSONObject.put()* *TelephonyManager.getSubscriberId() → JSONObject.put()* NO_CATEGORY → FILE: *JarFile.getInputStream() → FileOutputStream.write()* NO_CATEGORY → LOG: *File.getName() → Log.i()* *JSONObject.getInt() → Log.i()* *JSONObject.getString() → Log.i()* *Throwable.getStackTrace() → Log.i()* *Throwable.getMessage() → Log.i()* *Settings$Secure.getString() → Log.i()*	324c2630...28ea.apk	NO_CATEGORY → NO_CATEGORY: *File.getAbsolutePath() → String.substring()* *HashMap.get() → HashMap.put()* *HashMap.get() → String.substring()* *HashMap.get() → String.startsWith()* *File.getAbsolutePath() → HashMap.put()*
Fusob	960b722a...ab4c.apk	NO_CATEGORY → NO_CATEGORY: *Camera.getParameters() → Camera.setParameters()* *Settings$Secure.getString() → JSONObject.put()* *ContentResolver.query() → JSONObject.put()* *Intent.getExtras() → Intent.putExtras()* *Thread.getName() → JSONObject.put()* *Camera.getParameters() → Camera$Parameters.setRotation()* *Intent.getAction() → Intent.setAction()* *Log.getStackTraceString() → JSONObject.put()* *PendingIntent.getBroadcast() → AlarmManager.setRepeating()* *PendingIntent.getBroadcast() → AlarmManager.set()* *WebView.getSettings() → WebSettings.setJavaScriptEnabled()* NETWORK_INFORMATION → NO_CATEGORY: *TelephonyManager.getSimOperatorName() → JSONObject.put()* UNIQUE_IDENTIFIER → NO_CATEGORY: *TelephonyManager.getLine1Number() → JSONObject.put()* *TelephonyManager.getDeviceId() → JSONObject.put()* *TelephonyManager.getSubscriberId() → JSONObject.put()*	cdf394ce...c33a.apk	NO_CATEGORY → NO_CATEGORY: *Class.getConstructor() → OutputStream.write()* *Class.getMethod() → AssetManager.open()* *File.getAbsolutePath() → Field.set()* *Field.get() → Field.set()* *Class.getMethod() → Field.set()* *Class.getDeclaredMethod() → Field.set()* *Class.getName() → String.substring()*

the most expensive steps in our method. However, these complexities are negligible due to the powerful machines which are available nowadays, and, also, the detailed insights this behavioral labeling provides. Also, these complexities show that applying the same labeling scheme on new

Table 4.11: Average amount of time and memory consumed in each step of our clustering approach per application.

Dataset	Flow Extraction (FlowDroid)		Pattern Extraction (SPMF)		Clustering (k-means++)		Overall	
	Time (.sec)	Memory (MB)	Time (.sec)	Memory (MB)	Time (.sec)	Memory (MB)	Time (.sec)	Memory (MB)
Drebin [76]	45.23	1031.85	292.43	7.60	1.23	163.85	338.89	1203.3
AMD [79]	100.75	1181.36	432.60	8.90	2.19	247.06	535.54	1437.42

datasets is not a challenging task despite the fact that it is consisted of different preprocessing steps, including flow extraction, pattern extracting and clustering.

We had some valuable findings comparing the results we got from applying our labeling method on the recently released AMD dataset (with apps from 2010 to 2016) and the older Drebin dataset (with apps from 2010 to 2012). First, we observed an increase in the average number of information flows. In particular, apps have 41 unique information flows on average in AMD dataset as compared to an average of 26 information flows per app in Drebin dataset. Second, we found 36 new pairs of source and sink SuSi categories of information flows in the applications of AMD dataset. As flows can reveal how apps treat with sensitive user's or smartphone's data, we believe this shows that new and diverse behavioral profiles have been evolved in recent years. Third, information flows leaking geographical location information are 0.62% higher in the apps of AMD dataset comparing with the ones in Drebin dataset. On the contrary, flows leaking unique identifier information and network information are 2.66% and 2.27% higher in Drebin dataset. Moreover, leaking sensitive information through sending text messages and the network are 2.64% and 1.74% more common in Drebin dataset respectively though logging captured information is more popular in AMD dataset by 2.4%. Last but not least, patterns of flows have increased in both number and length in AMD dataset.

4.4 Discussion

We next discuss some potential limitations of our approach to produce a behavioral characterization of Android malware.

4.4.1 Accuracy

The accuracy of our approach is critically dependent on the accurate identification of real information flows appearing in each application, which are extracted using FLOWDROID. However, advanced obfuscation techniques used in special types of malware can bypass static analysis tools and, thus, flows containing in these apps might be missed. Also, using reflection makes static taint analysis tools to miss some flows, particularly if methods are invoked dynamically at runtime. We are not aware of any study measuring the popularity of reflection in current malware samples and, therefore, we cannot measure the extent of this limitation.

Moreover, our approach has some limitations in characterizing apps which use collusion attacks, since most static taint analysis tools do not consider information flows across apps (i.e., when the source is located in one app and the data is passed on to another app that access the sink). Here, malicious behavior which is delivered through collusion is missed.

4.4.2 Datasets

Our proposal and its experimental results might be affected by the number and variety of malware samples in our datasets. While the exact coverage of our datasets is not known, we believe it is fairly representative as it contains different types of malicious apps. Our study includes *Drebin*, which extends the widely used *Malgenome* dataset and has been consistently used by most works in the Android malware area in the last years. We also included in our study one of the newest datasets of Android malware (AMD), which reinforced our main conclusions since the conclusions extracted from both datasets proved to be similar. The limiting factor here is the extraction of information flows, which requires a substantial amount of computational resources and fails for a fraction of the apps analyzed. However, these limitations are not critical for two main reasons. First, powerful computational resources and memory are both available nowadays. Second, our scheme is not coupled with a particular flow extraction tool and can benefit from further advancements in this area.

4.4.3 Repackaged Apps

Repackaged apps might also affect the characterization scheme proposed here. Most of these variants are composed of malicious payloads contained within popular legitimate apps known as carriers. Since malicious payloads have relatively smaller code sizes [195] than carriers, most of the flows extracted from each application for behavioral-based characterization may come from the benign carrier of the repackaged app and not the malicious payload. If this happens, a flow-based characterization scheme cannot reflect the real malicious intent of repackaged apps precisely. Furthermore, it is not clearly known whether extracted flows are coming from the carrier or the malicious payload, which can affect the proposed scheme as well. If the majority of flows extracted from the repackaged app belong to the carrier, the sample would resemble the benign legitimate application that has been used for this repackaged version. Contrarily, if most of the flows belong to the payload, this repackaged sample would be identified as a completely different application compared to the incorporated legitimate application.

4.5 Related Work

This section discusses four main related areas to this work, including information flow analysis in Android, info-flow based Android malware detection, pattern mining in Android application analysis, and, ultimately, malware characterization and classification.

4.5.1 Information Flow Analysis

Information flow analysis is a valuable technique to track how information is transferred within the system [9]. Although information flow analysis has been used in analyzing both PC and mobile malware, information flow analysis in Android is not a trivial task for various reasons [196]. Firstly, Android components can be executed in any arbitrary order depending on the user interactions, which makes the flow graph fragmented. Secondly, there is not a single entry point in Android applications [169], which makes information flow analysis more complex than for traditional PC programs.

Taint analysis is a commonly used approach to track information flows which can be performed either statically or dynamically. Generally, it is a process through which taint labels are assigned to sensitive data when they leave designated source methods (e.g., *getDeviceId()*), and, afterwards, different procedures are performed while these data reach specific sink methods (e.g., *sendTextMessage()*). Both of these taint analysis techniques have a number of limitations despite their advantages.

Static taint analysis tools, including DroidSafe [93], FlowDroid [92], FlowMine [94], LeakMiner [96] and CHEX [95] are applied before runtime and require a considerable amount of resources (CPU and RAM memory). They can be bypassed using advanced (or sometimes simple) obfuscation techniques [197]. Moreover, they are imprecise and have a high false positive rate as they do not consider runtime behavior of applications [198]. Finally, they suffer from scalability issues as they need to traverse the whole call-graph of applications [199].

On the other hand, dynamic taint analysis tools such as DroidScope [23] and TaintDroid [98] are applied at run-time or in a simulated environment. These tools may miss some flows which are not exercised explicitly at runtime [157]. Furthermore, they impose a high computational overhead and can be bypassed using novel strategies for simulator detection which has been adopted recently by malware developers [200].

4.5.2 Android Malware Detection Using InfoFlows

Due to the fine granularity of information flows, they have been widely used for two main purposes in the area of Android malware analysis. The first group of works have made use of information flows in Android malware detection, while the second group of works have assessed apps security by producing risk scores based on information flows.

DroidSieve [201] is a recently proposed tool that relies on sensitive information flows extracted from static taint analysis. It also extracts some additional features, including permissions, code structures, and the set of invoked components to create a fingerprint (i.e., a set of features) for each malware sample. DroidAPIMiner [129] uses the information in critical API calls and their parameters to distinguish malware from benign applications. DroidMat [202] is a malware detection system which works based

on tracing API calls and extracting some extra information which is obtained from the *Manifest.xml* file (e.g., requested permissions and intents).

TriFlow [185] is a risk scoring system for Android apps using information flows. In the first step, it assigns a degree of maliciousness to different information flows based on their frequency in malware samples and their rarity in benign apps. Next, it calculates a risk score for each application based on the occurrence probabilities of information flows and their amount of maliciousness.

4.5.3 Pattern Mining in Android Apps

Pattern mining has been frequently used in recent works related to Android app analysis. One of the main areas in which pattern mining algorithms have been applied is for usage prediction in smartphones [119] [120]. Most of the works in this area have focused on mining behavioral patterns (or profiles) from Android applications using different features. For instance, a mining algorithm has been proposed in [121] to extract temporal API usage patterns from client programs in order to help developers having a precise and complete understanding of the current libraries. A similar work [122] extracts time-constrained sequential patterns using mining algorithms to identify application usage patterns on smartphones. ApMiner [123] relies on association rule mining of android apps in the market to identify co-occurrences of permissions and API methods. Based on this, it recommends specific permissions which need to be added when developers use special API methods in their programs.

Another area in which pattern mining algorithms have been adopted is malware characterization. In a very recent work [124], information on apps descriptions, together with sensitive data flow signatures, have been used to characterize 3,691 malicious and 1,612 benign applications. Here, a model is used initially to cluster apps based on the information appearing in their descriptions. The information gain ratio is then used to generate a topic-specific flow signature for each topic. These flows are a list of patterns that appear in the apps of a corresponding topic in which each pattern is assigned a gain value indicating its power to discriminate malware from benign applications.

4.5.4 Malware Characterization and Classification

Several works have addressed the problem of how characterize and classify both desktop and Android malware, as this contributes to a better understanding of the current behavior of malware in the wild as well as identifying new malware specimens.

The majority of works in the area of desktop malware classification rely on different features extracted from binaries. An early attempt in this area compared the Control Flow Graph (CFG) of unknown malware samples with previously known ones [203]. CFGs provide precise information of the structure of the program in general and its sub-routines, as it reveal the paths which are induced by different instructions such as conditional branches. Other works have explored the idea of classifying malware by using a variety of other features, such as (function) call graphs [204, 205].

Most of the methods proposed for Android malware classification rely on static features, which are obtained either from the app package or through static analysis. The very first work in this category dates back to 2012 [39], where the authors proposed a method to systematically characterize Android malware using three main features, including the way through which they are activated, their installation method, and the nature of their malicious payloads. A more recent work [206] addresses Android malware characterization by studying the behavior of apps' malicious payloads (known as riders) after removing irrelevant code segments using differential analysis. In another relevant work [207], intent actions, API Android packages, and sensitive API calls are used to characterize Android malware. Finally, [208] proposes a method to classify Android malware based on the frequent subgraphs which are extracted from the CFG of applications. In addition to all these works, a few tools like androsim [209] and dexid [210] can be considered to check if two apps share common methods or classes, and to classify them based on these similarities. Very few works have proposed characterization schemes based on dynamic features. One recent example is [211], in which Android malware is profiled in a sandbox and their invoked API calls are used to perform similarity analysis and to group them into various categories.

Despite this effort in characterizing Android malware, most of the works in this area–in particular, those relying on static features–are unable to produce an easy-to-interpret description of the sample's real behavior. This is critically important as the number of Android malware variants of already

known families have raised significantly in comparison to apps from new malware families [184, 195].

4.6 Conclusions

In this chapter, we have discussed the problem of how family labels in Android malware relate to the behavior of their samples. Our approach relies on modeling apps through their information flows and then characterizing behavior by patterns of such flows. We have also conducted a cluster analysis to identify groups of apps whose patterns of information flows are similar. Our experimental results show that the notion of Android malware family does not necessarily imply that all its samples behave similarly, and also that different malware families sometimes behave identically.

Our future work includes two main directions. First, we will explore the application of a similar approach to characterize other types of malicious binaries, such as traditional desktop malware. To do so, we intend to leverage tools like Panorama [212], a system to capture information flows for malware detection. Second, the same method can be used to characterize benign applications based on their information flows and obtain behavioral profiles. This can help in identifying and labeling legitimate applications that, however, involve sensitive information flows and might potentially involve security and/or privacy violations.

5

AndrODet: An Adaptive Android Obfuscation Detector

5.1 Introduction

The widespread usage of smartphones in various security-sensitive operations in recent years, such as bank transactions and online payments [213], requires that the security of these platforms must be improved. This affects particularly to smartphones hosting Android applications, as they have the biggest world-wide market share [214]. More specifically, in recent years where re-packaging popular smartphone banking applications has raised in number [215], hardening apps against reverse engineering has become increasingly important.

Source code is an important intellectual property for both legitimate software developers and malware writers; specifically, in Android operating system where the applications can be easily decompiled for automated code analysis or visual inspection. In the legitimate context, obfuscation prevents the competitors from cloning or copying the source code with little effort and just by adding very few extra features, while in a non-legitimate context, it hides the apps' semantics from analysts by increasing the cost of reverse engineering and decompilation.

Obfuscation has been vastly applied to both malware and benign Android applications in the last years [7]. In particular, three types of obfuscation have been used, including identifier renaming, string encryption, and control flow obfuscation mainly because they are either available in free obfuscators or in the trial versions of commercial obfuscators. Also, they create a satisfactory level of confusion in the app's source code. Based on

previous researches [7], malware writers prefer to make use of more complex renaming policies than legitimate software developers. Also, string encryption is more popular in malware than in benign apps. Finally, although control-flow obfuscation is only offered by few commercial obfuscators, its prevalence and detection has not been studied before.

Prevalent usage of obfuscation in Android malware has also cast doubt on the reliability of most Android malware analysis tools [19] [104], and, in particular, static ones. The majority of these tools rely upon some static features which are obtained from the source code and are severely impacted by little transformations in the source code [19]. Consequently, they are not resilient to transformation attacks. Also, obfuscation has turned out to be a new barrier to protect Android users [8], and, therefore, detecting obfuscation is critical in understanding the underlying semantics of malware specimens.

Previous works leverage on batch learning systems to detect obfuscation. Thus, after extracting a set of features from the apps pooled as training set, a system is trained to detect one or more types of obfuscation [7] [20]. While these systems offer promising accuracy rates, they do have a major drawback. Systems which work based on batch learning do not necessarily remain effective over time - when new applications appear or when novel obfuscation techniques are proposed. Thus, they must be eventually re-trained with the updated dataset. This task is not feasible in a setting where apps are developed and introduced constantly (as it currently happens in both Android malware and benign apps). Also, most of the recent works have tried to detect trivial types of obfuscation on a small dataset of apps. Finally, advanced obfuscation techniques such as control flow obfuscation has not been addressed based on a representative recent malware dataset [20].

To overcome these limitations, in this chapter, we explore the use of online learning algorithms through Data Stream Mining (henceforth DSM) [216]. DSM can be seen as an adaptation of traditional machine learning methods so as to be suitable for streams of elements. Remarkably, DSM approaches do not need to be re-trained, as they continuously learn from the input samples. Leveraging DSM, we aim to detect basic forms of obfuscation (particularly, identifier renaming and string encryption), as well as the non-trivial control flow obfuscation. To assess our approach, we

consider a dataset of 34962 samples from both malware and benign applications.

We propose AndrODet, an online learning system to detect three common types of obfuscation techniques in Android applications, known as identifier renaming, string encryption, and control flow obfuscation. All of these obfuscation techniques are detected based on some static, quick-to-obtain features extracted from the Dalvik executable bytecode of applications. AndrODet is modular, meaning that there is a separate embedded module within the system to detect each type of obfuscation, and each of these modules are trained separately.

AndrODet has been implemented in python and tested on a combination of malware and benign samples. The former set of apps are collected from a recently released and carefully-labeled malware dataset, called AMD [79], while the latter are obtained by crawling the popular open-source repository of benign apps known as F-Droid [84]. We have also compared our results with state-of-the-art batch learning algorithms by leveraging Auto Tune Models (ATM) [217], a system developed for hyper-parameter tuning of batch learning algorithms and classification using a variety of algorithms from this kind.

Experimental results show that online learning algorithms can detect three popular types of obfuscation techniques in Android applications with high accuracy. In addition, they can save significant amount of time and memory as compared to batch learning algorithms.

In short, the main contributions of this chapter are as follows:

- We propose AndrODet, a modular online learning mechanism to detect identifier renaming, string encryption, and control flow obfuscation in Android applications. To allow future works benefit from this research, we make our tool publicly available at:

  https://github.com/OMirzaei/AndrODet

- As AndrODet is based on DSM techniques, there is no need to retrain the system from scratch. Thus, we compare the effectiveness of our system with machine learning algorithms working based on batch learning. To do this, we leverage MOA [216] and add some extra features to this tool for hyper-parameter tuning which will be used later for classification. This enables us to have a fair comparison between the results obtained from online learning algorithms

using MOA and the ones which are obtained from batch learning methods using ATM.

- ANDRODET is able to deal with multidex Android applications. Our system looks for all classes.dex files in different directories and extracts its features from all of them.
- We assess the efficiency of our tool with AMD [79] and PraGuard [78]. Both datasets, with more than 24k apps in total, contain ground truth for apps which are obfuscated by identifier renaming and string encryption techniques. Moreover, to create ground truth for control flow obfuscated apps which was previously lacking, we have leveraged a well-known obfuscator known as Allatori [218] and have obfuscated all the samples of F-Droid [84], a free and open source Android applications repository. We aim at publicly releasing the latter set of apps to foster further research in this direction.

The remainder of this chapter is as follows. Section 5.2 introduces common obfuscation techniques which are applied to Android malware in order to evade detection. Section 5.3 describes the proposed system. Evaluation results are presented in Section 5.4 followed by a discussion in Section 5.5. Section 5.6 surveys some related works, and, finally, Section 5.7 concludes the chapter and presents future research directions.

5.2 Obfuscation in Android

Obfuscation is commonly used to protect software against reverse engineering, thus making the software harder to understand [219]. There are multiple obfuscation techniques [220]. In this work we focus on three well-known obfuscation techniques that are commonly applied to Android applications, namely identifier renaming, string encryption, and control flow obfuscation [220] [221].

A common practice in programming is to choose meaningful names for identifiers (i.e, variables, class and method names, etc.) to increase the code readability. This will helps in identifying and fixing bugs or adding extra features later, as understanding the semantics of code with meaningful identifiers is much simpler. However, malware writers try to choose either meaningless names for their identifiers or else use obfuscators in order to garble the key identifiers used in their source code. Obfuscators use a variety of methods to rename key identifiers of an application either at the

source code level or directly in the .dex files. An obfuscated identifier can be often told apart visually from a non-obfuscated one because its name is meaningless. For example, a common renaming strategy is to choose random short strings in lexicographic order, e.g., 'a', 'b', 'aa', 'ab', 'ac', etc., usually with lengths less than 3 depending on the number of identifiers. A second strategy is to leverage the overloading feature of Java through excessive overloading and map irrelevant identifier names to the original ones.

By doing so, reverse engineers need to put much more effort into understanding the hidden semantics of code when critical information such as method names are obscured. Based on a recent study [7], the prevalence of identifier renaming is slightly less in malware than in benign apps from third-party markets. Also, malware authors tend to use more complex renaming policies, such as using special characters (e.g., encoded in Unicode), which creates challenges for systems which are developed to detect this type of obfuscation.

Constant strings can also leak sensitive and private source code information. Thus, they are encrypted in different ways to prevent a convenient reverse analysis of applications. The most simplest way to encrypt encryption is through an XOR operation. However, standard cryptographic algorithms can be applied, including AES or DES [222]. Also, secret keys can be defined (or either changed) dynamically to apply more advanced types of string obfuscation, which is almost impossible to be handled by static analysis tools. Studies show that string encryption is more popular in malware and nearly all benign apps do not make use of this type of obfuscation.

Control flow obfuscation hinders static analysis by changing the logical flow of the program through modifications in its Control Flow Graph (CFG). Typical techniques from this category try to expand or flatten the CFG in order to increase the cost of reverse engineering of applications. Common ways to do this include injecting dead (or irrelevant) code, extending loop conditions, adding redundant operations, parallelizing code, re-ordering statements, loops, and inserting opaque predicates. The majority of these approaches affect some properties of the CFG, such as the number of nodes and branches. Based on recent observations, control flow obfuscation is not widely used, and it is only offered by a few number of commercial obfuscators such as Allatori [218] and DashO [223].

5.3 Approach

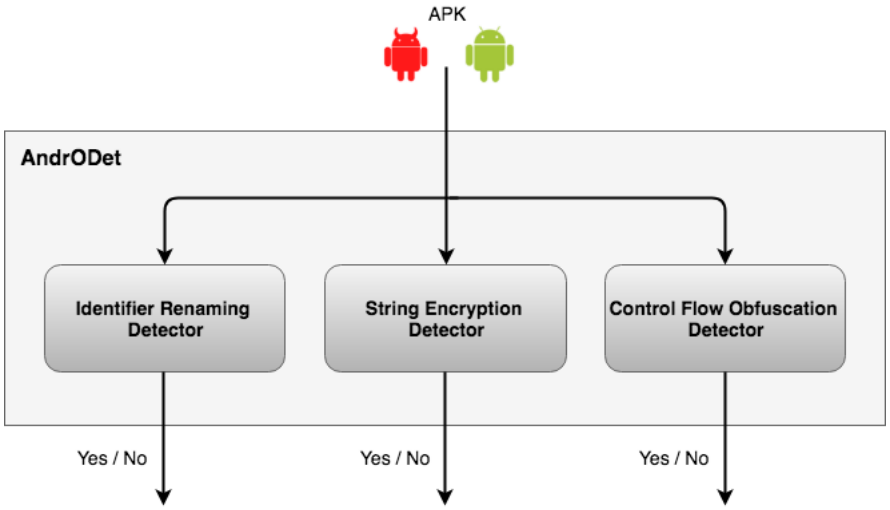
This section presents our approach to detect three types of obfuscation techniques in Android applications. A general overview of the system is proposed in Section 5.3.1. Then, primary goals are clearly defined in Section 5.3.2. In Section 5.3.3, we describe all the details related to the datasets which are used in this work. The set of all features considered for our detectors and possible feature selection algorithms are discussed in Section 5.3.4. Finally, classification algorithms chosen for our online learning system and their hyper parameter tuning are presented in Section 5.3.5.

5.3.1 Overview

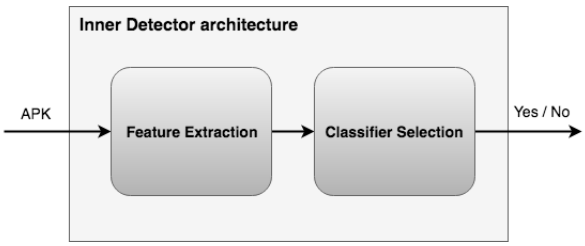
ANDRODET is an online learning system which is developed to detect three main types of obfuscation in Android applications, namely identifier renaming, string encryption, and control flow obfuscation. Also, it can detect obfuscation in Multidex Android applications. Android RunTime (ART) which is used in Android 5.0 (API level 21) and higher supports loading multiple Dalvik EXecutable (DEX) files from APK files. It then performs pre-compilation at install time and scans for all classes.dex files to compile them into a single .oat file. This feature enables applications to distribute their code into several .dex files. Specific Android malware variants have also been observed which load their malicious .dex file from a secondary directory (e.g., assets directory) [224] [225]. ANDRODET searches for all classes.dex files in different directories and extracts its features from all of them.

The proposed system is modular, i.e., there is an embedded module (binary classifier) to detect each type of obfuscation as shown in Figure 5.1a. Using a modular architecture has three main advantages. First, it reduces feature overlap, and, thus, improves the precision accuracy. Second, the system can be easily updated with a new set of features for each module based on variations in obfuscation techniques. Third, different learning algorithms can be used for each module based on the nature of the input data.

To label new unseen apps, all required features are extracted by each module and a feature vector is created at the first step, as depicted in Fig-



(a) ANDRODET global structure.



(b) Inner structure of each detector module.

Figure 5.1: ANDRODET architecture.

ure 5.1b. A binary classifier is then chosen to decide whether or not the app is obfuscated. These classifiers are trained incrementally using online learning algorithms while labeling new applications.

5.3.2 Goals

ANDRODET is intended to achieve the following main goals:

- **Rapidity**. The system must be able to work in a reduced amount of time.
- **Readiness**. The system must be ready to work with moderate training requirements.
- **Accuracy**. The system must accurately identify the type of obfuscation that has been applied.
- **Scalability**. The system must be able to cope with a large number of applications using a moderate amount of resources.

5.3.3 Dataset Description

Our dataset is formed by both malware and benign applications, and contains ground truth for all of the obfuscation techniques considered in this work. We have built up the ground truth for identifier renaming and string encryption obfuscation techniques by leveraging the AMD dataset [79], a recently released Android malware dataset with apps from 71 families ranging from 2010 to 2016 (Table 5.1). This dataset is formed by 24,553 applications that are labeled based on a number of behavioral criteria, including the presence of different anti-analysis techniques (e.g., identifier renaming or string encryption) in the apps of each family of one particular variety. To have a fair and balanced ratio of obfuscated and non-obfuscated samples, we have selected the same number of apps for each type, some of which were obfuscated using more than one technique.

In order to create a dataset of Android apps for control flow obfuscation technique, 1,380 applications were downloaded from the F-Droid market [84]. Both the compiled app package (APK file) and their Java source code are available in the market. Therefore, they are used as the ground truth for non-obfuscated apps. Also, to gather the same number of control flow obfuscated apps, we apply Allatori [218] over 1,380 apps selected randomly from AMD dataset. These apps are control flow obfuscated to the maximum level[1]. According to Allatori documentation, this level of obfuscation makes the apps bigger in size and a little bit slower as it uses all types of control flow obfuscation techniques. We finally choose 80% of this repository (2208 apps) to assess the accuracy of control flow obfuscation detection module, and we leave the remaining 20% (552 apps) to test its efficiency over unseen applications. The ratio of obfuscated and non-obfuscated samples is again equal in both portions.

Finally, we have used an additional released dataset, known as PraGuard [78] to evaluate the performances of our identifier renaming and string encryption detector modules over unseen applications. This dataset is composed of 10,479 samples, obtained by obfuscating the MalGenome [39] and the Contagio Minidump [77] datasets with seven different obfuscation techniques. It is worth mentioning that during our feature extraction process, we found that some apps cannot be disassembled properly with dexdump, and, thus, we have discarded them from our datasets.

[1] http://www.allatori.com/doc.html

Table 5.1: Number of apps per obfuscation technique

Dataset	Identifier Renaming		String Encryption		Control Flow Obf.		Global	
	Obf	**Non-obf**	**Obf**	**Non-obf**	**Obf**	**Non-obf**	**Obf**	**Non-obf**
F-Droid	0	0	0	0	0	1,380	0	1,380
AMD	5,992	5,992	7,119	7,119	1,380	0	14,491	13,111
PraGuard	1,495	1,495	1,495	1,495	0	0	2,990	2,990
Total	**7,999**	**7,999**	**8,614**	**8,614**	**1,380**	**1,380**	**17,481**	**17,481**

Table 5.2: Set of all features considered for each detector module

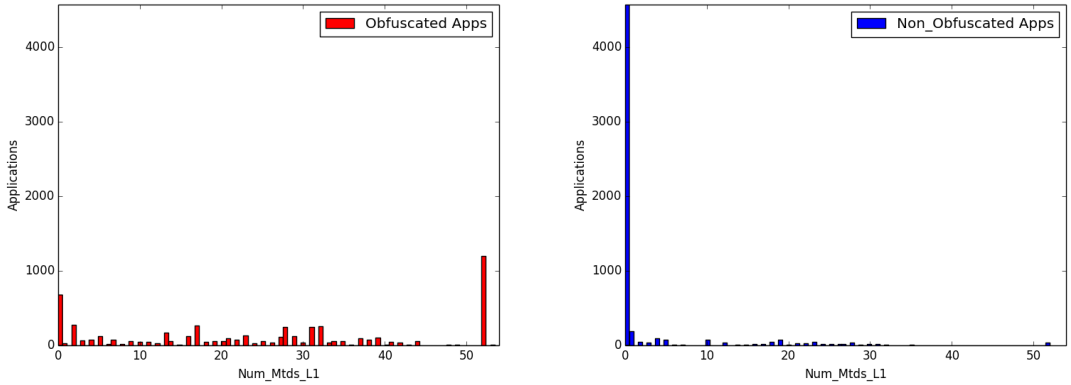
Identifier Renaming	String Encryption	Control Flow Obfuscation
Avg_Wordsize_Flds	Avg_Entropy	Num_Nodes
Avg_Distances_Flds	Avg_Wordsize	Num_Sinks
Num_Flds_L1	Avg_Length	Num_Edges
Num_Flds_L2	Avg_Num_Equals	Num_Goto/LOC
Num_Flds_L3	Avg_Num_Dashes	Num_NOP/LOC
Avg_Wordsize_Mtds	Avg_Num_Slashes	LOC
Avg_Distances_Mtds	Avg_Num_Pluses	File_Size
Num_Mtds_L1	Avg_Sum_RepChars	
Num_Mtds_L2		
Num_Mtds_L3		
Avg_Wordsize_Cls		
Avg_Distances_Cls		
Num_Cls_L1		
Num_Cls_L2		
Num_Cls_L3		

5.3.4 Feature Extraction and Feature Selection

The first important decision to make in learning-based systems is to choose the set of features that will be used to label (or predict) new unseen instances. Once they are defined, analysts may decide to apply feature selection algorithms to discard those features that are not relevant despite the initial assumption, or those with a low variance among all instances. In our case, we aim to identify a set of features that, apart from being useful for the prediction task, can be rapidly extracted from the applications. Thus, we simply parse the Dalvik bytecode (recall Section 2.2) of each app using dexdump [26] to find the majority of features. Table 5.2 shows the set of all features considered. In addition, the distributions of all features extracted from all apps in our dataset are included in the Appendix for further analysis. In what follows, we describe them per module in more detail.

5.3.4.1 Features for Identifier Renaming Detection

To detect identifier renaming, we extract 5 different features from the key identifiers of Dalvik bytecode, including fields, methods, and classes. The set of features considered here are the average wordsize (in bytes), the



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.2: Distribution of methods with length 1 in obfuscated (a) and non-obfuscated (b) apps.

average distance of consecutive extracted identifiers, and the number of identifiers with length 1, 2 and 3. To compute the distance between two identifiers, we first represent each string as a vector of natural numbers, where each component is given by the corresponding byte in the string. If they are not of the same length, the shorter identifier is right-padded by blank spaces. After this, the d_1 distance between both vectors is computed:

$$d_1(A, B) = \sum_{i=1}^{n} |a_i - b_i|, \tag{5.1}$$

where $A = (a_1, \ldots, a_n)$ and $B = (b_1, \ldots, b_n)$ are the byte-level representations of both strings. Since we operate at the byte level, we refer to this as the ASCII distance of the two identifiers.

The rationale for the ASCII distance is the following. When using renaming, identifiers are normally replaced by repetitive or random sequences of characters in the English alphabet in Android benign apps, and special characters (encoded in Unicode) in malware samples [7]. Thus, consecutive extracted identifiers in ID-renamed malware samples usually have a small ASCII distance compared to the ones in benign apps, as shown in Tables 5.3 and 5.4. Moreover, based on our observations (Fig. 5.2), the number of identifiers with lengths lower than 3 were much more frequent in obfuscated samples than in benign apps, which provides additional support to our logic to choose this set of features for identifier renaming detection.

Table 5.3: Examples of identifiers extracted from an obfuscated malware sample in the Obad family.

App MD5 = f7be25e4f19a3a82d2e206de8ac979c8		
List of fields	**List of methods**	**List of classes**
cOIcOOo	ocCCIII	oIoICCOc
lOocoOI	onOpen	AdminReceiver
loOoOlOI	onUpgrade	cOoOlCO
oclCIII	OoCOocll	lOocoOI
OoCOocll	OOIIIcCc	OlCCcll
OoclOCIo	onCreate	OcIcoOlc
OllCCCco	cClccOlc	OoCOocll
CICCCcCI	CcOCoIcO	OOIIIcCc
occcclc	oIOocIIo	OoclOCIo
oOCCOOI	CoOOoOo	ClOIIolc
oCOllOO	oIlclclc	CICCCcCI
ClOIIolc	lCclCcoC	olcCIIC

Table 5.4: Examples of identifiers extracted from a non-obfuscated malware sample in the Univert family.

App MD5 = dadba61b42e3129dcbb2c37ba7177290		
List of fields	**List of methods**	**List of classes**
mBigLargeIcon	getItemId	KeyEventCompatEclair
mParentFragment	isSingleShare	ViewPager
mSetIndicatorInfo	performPause	ContextCompat
EDGE_ALL	makeMainSelectorActivity	NotificationCompat
mPendingBroadcasts	setDrawerShadow	ParcelableCompat
TRANSIT_NONE	getCallingPackage	ScrollerCompat
mHandler	getConstantState	TransportPerformer
mTaskInvoked	setUserVisibleHint	PagerTitleStrip
mNumOp	setMenuVisibility	TimeUtils
PRIORITY_DEFAULT	setOverScrollMode	BackStackRecord
ACTIVITY_CREATED	dismissAllowingStateLoss	FileProvider
children	dataToString	SupportMenu

5.3.4.2 Features for String Encryption Detection

For string encryption detection, we considered 29 different features at the beginning, all of which were obtained from the app bytecode. The set of initial features we considered included: the average entropy, the average wordsize, the average length, the average number of equals ('='), the average number of dashes ('-'), the average number of slashes ('/'), the average number of pluses ('+'), the average sum of repetitive characters which are appear more than once in a string, and the frequency of 21 different special characters, including underlines and spaces. However, we were left

Table 5.5: A snapshot of constant strings extracted from obfuscated malware in the Kyview and Triada families.

App MD5: 9f973194e1d2db2c8d37571b1b8afa49, Family: Kyview
AES
AES/CBC/PKCS5Padding
ARuhFl7nBw/97YxsDjOCIqF0d9D2SpkzcWN42U/KR6Q=
KXbn1K9Cz2ZgeOTJa+Veo9TtqgqFQ49etShsU9z+UAP37syBIxS/qy9gK8yB2kKw
cbSAmn5ZqTUlLC/bgOZkEzXGEOY21uWifgdKJs9yk7A=
XONjlhr7f5+v7VYE2sRnrybwgpe9YIOqpcEHDUiel7EzNqAyl0RSFuWdEz2ratN+
LbZjxcpsz6RheqLbO48YwKTUVh9wQrFoY7gJK2jAZFI=
/XHxH5XHwv8SxKlJV4XyYOIB7MuqmSwqMacPj1bbgbS8IA8tETEArriXswHCehFP
Jil+B/2MHKx+6dpy/2xm493DojzmiB3wB5+eGz7hPDU=

App MD5: a19f784807c3249837135de9b1a43fdf, Family: Triada
Sw4QQ1hFGFJJF1UWDwN1dnYKVQQGJAJDWwMUYkZVEUYHQg==
Wg4WQ2hRRkNySV8BOUNVX1U=
UQ4IGU5EGEZYF1UWDwN1dnYKVQQGJAJDWwMUYkZVEUYHQg==
VxkRaFZCUWxdS1sWOUtdXlU8QwAPAw==

with only 8 features after applying feature engineering techniques, namely the average entropy, the average wordsize, the average length of strings, the average number of equals, dashes, slashes, and pluses, and, finally, the average sum of repetitive characters. This was done using a tree-based feature selection algorithm which scores features based on their importance and discards irrelevant ones [226].

We chose this set of features by visually analyzing a number of strings from both obfuscated and non-obfuscated samples. Critical constant strings in Android malware are normally encrypted by either AES or DES encryption algorithms [79]. Also, they are commonly encoded using Base64 scheme. These block cipher algorithms, depending on the mode which is adopted, require the input string to be an exact multiple of the block size. If the string to be encrypted is not an exact multiple, it is padded before encrypting by adding a padding string (or a pad byte). In our studies, we observed many strings in obfuscated samples which were padded by using '=' or '==' strings (Table 5.5). Furthermore, equal signs, dashes, slashes, and plus signs are observed mostly in obfuscated strings than in non-obfuscated ones.

5.3.4.3 Features for Control Flow Obfuscation Detection

Finally, to detect control flow obfuscation, features are extracted from both Dalvik bytecode and the CFG of applications. Seven different features are extracted here: the number of nodes; the number of sinks (i.e. nodes with

an outdegree = 0); the number of edges from the CFG of each application; the number of goto instructions per line of code; the number of NOP instructions per line of code; and the total number of lines of code from the app's bytecode. Additionally, the app's file size is considered because some advanced types of Android malware pack their native code in the resource or assets directories and decrypt them at runtime using a decryption stub [31] [151]. So, this feature compensates for the limitations of dexdump in accurately measuring the lines of code from sophisticated Android malware specimens.

Although features for control flow obfuscation detection are extracted from both bytecode and the CFG of apps, we had the intuition that some code features may overlap with others extracted from CFG. For instance, goto instructions simply add more branches to the CFG, and, thus, increase the in-degree or out-degree of some nodes. However, extracting features from both bytecode and the CFG guarantees that no features will be missed due to the limitations that may exist in Android reverse engineering tools.

5.3.5 Classification Algorithms and Hyper-Parameter Tuning

The second critical decision in learning-based systems is to choose an appropriate classifier to label unseen samples. Additionally, most of these classifiers have various parameters which have significant impacts on their performance. They are commonly known as classifiers' hyper-parameters which need to be set wisely based on the application context. One simple example is the number of neighbors (k) in the famous k-Nearest Neighbor (or kNN) learning algorithm [227].

Three strategies are usually adopted to tune classifiers' hyper-parameters [228]. In the first approach, all combinations of hyper-parameter values are tried in a greedy way to find the best possible set of combinations. In the second approach, all combinations are explored again but in a random fashion. The advantage of this method is that it may find the optimal solution faster than a greedy search. The third strategy is to use a random search but with a limited number of trials, which will make the algorithm even faster but does not guarantee finding the optimal set of combinations.

In ANDRODET, all classifiers update their models while observing new applications based on online learning algorithms. To do this, we have used a wide variety of algorithms provided by MOA, including Hoeffding Tree [229], Weighted Majority Algorithm [230], Leveraging Bag [231], LearnNSE [232], Stochastic Gradient Descent (SGD) [125], and Naive Bayes [233]. Moreover, we have extended this tool to enable us choosing the best possible hyper-parameters for the classifiers by developing a hyper-parameter tuning procedure. From the three discussed strategies, we have chosen limited random search, which gave us a satisfactory classification performance in a reasonable period of time.

5.4 Evaluation

This section presents the evaluation results. We first present the experimental settings. Then we evaluate the performance of each ANDRODET's detection module separately (Sections 5.4.2 - 5.4.4). Finally, we consider cases in which apps may be obfuscated with more than one technique (Section 5.4.5).

Additionally, we test the accuracy of our system on unseen apps (as discussed in Section 5.3.3) and compare the results with a similar system based on batch learning algorithms. We adopt the same strategy here, i.e., we initially test the performance of each module on unseen apps (Sections 5.4.6.1 - 5.4.6.3), and, then, we present the accuracy of system when apps may use a combination of obfuscation techniques (Sec. 5.4.6.4). We finally compare the performances of both systems in terms of time and memory usage in Section 5.4.7.

5.4.1 Experimental Settings

Experiments were carried out on an Ubuntu server with 15 processors and 24 GB of RAM. We use Massive Online Analysis (MOA) in its version as of February 2018 [113] to analyze the accuracy of ANDRODET. Also, to compare its efficiency with a similar system based on batch learning algorithms, we leverage the Auto-Tuned Models (ATM) tool [217], a recently proposed tool for machine learning and hyper-parameter tuning. We have selected various learning algorithms from this tool, namely kNN,

Support Vector Machines (SVMs) [234], decision trees [235], and random forests [236].

For online learning algorithms, we have used leveraging bag, and, from batch learning ones, we have finally selected SVM after observing the performances of classifiers. Moreover, to have a fair comparison, we first tune the hyper-parameters of classifiers (Figure 5.3) in both MOA and ATM following a limited random search strategy with 200 trials (known as budget in ATM). This helps us to obtain fairly well combination of parameters for each learning algorithm.

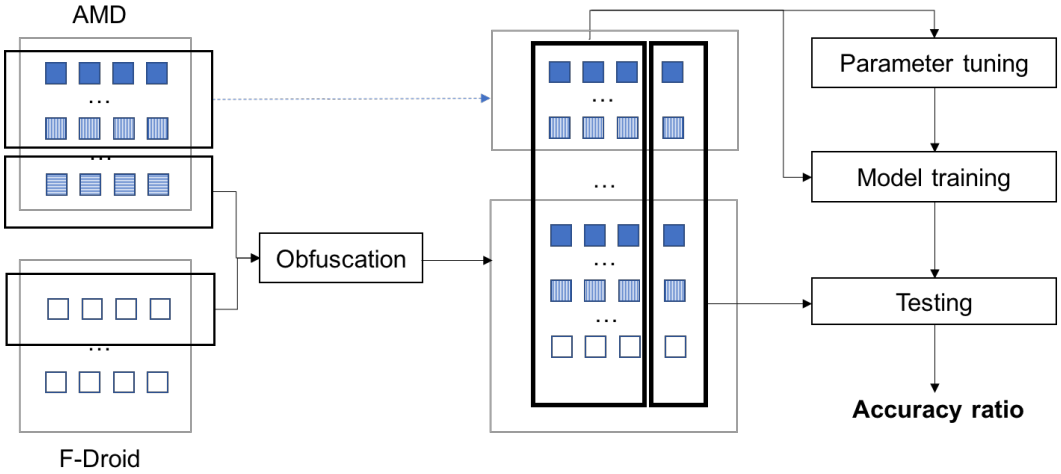


Figure 5.3: Data preparation (left) and the overall architecture of classification process (right), including parameter tuning, model training and testing. White squares: non-obfuscated apps; dark blue squares: apps with string encryption obfuscation; dashed blue squares: apps with ID renaming obfuscation.

5.4.2 Identifier Renaming Detection

We use the full AMD dataset in order to inspect how the accuracy of identifier renaming module evolves over time using the *EvaluatePrequential* class of MOA [237]. This class evaluates a classifier on a stream by testing, and, then, training with each sample in the sequence. Experimental results show that ANDRODET identifier renaming detection module is able to predict whether an app is obfuscated or not with a high accuracy immediately after observing few samples. As it is shown in Fig. 5.4a, the accuracy reaches around 71% after observing only 25 samples. Also, it

improves step by step by observing more samples from the dataset. Our module to detect identifier renaming obfuscation could achieve an average accuracy of 92.02% over the whole AMD dataset.

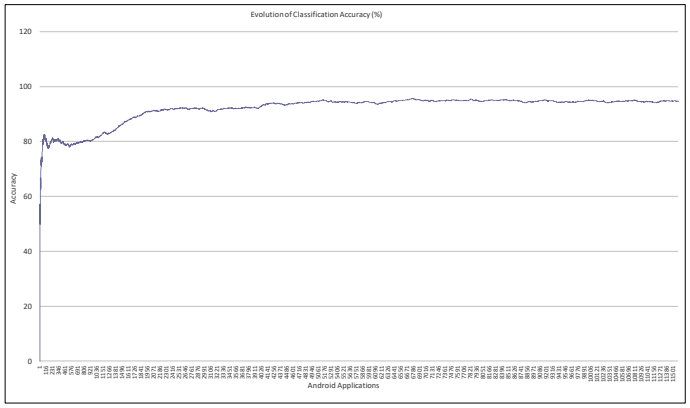
The number of samples correctly classified (TP) as obfuscated is 5758, and 631 samples were incorrectly classified (FP) as obfuscated (Table 5.6). One reason is that some obfuscators use a different strategy to rename key identifiers of malware samples such as using non-ASCII characters. The second reason is that non-obfuscated malware specimens do contain obfuscated identifiers as well in the majority of cases mainly because they import some classes from Android or Google libraries which are already obfuscated.

5.4.3 String Encryption Detection

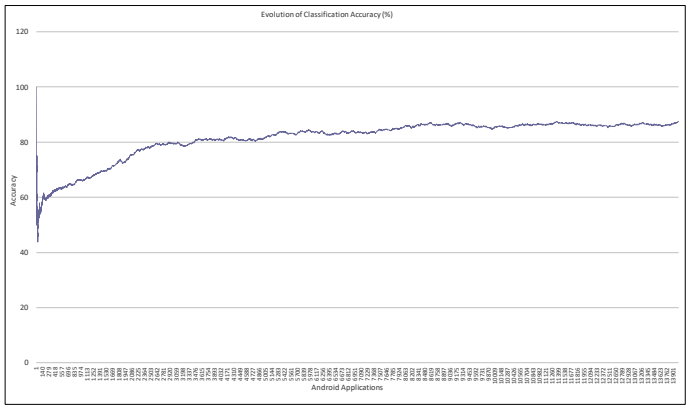
As malware samples use a wide range of cryptographic functions, classifying apps as either obfuscated or non-obfuscated is not straightforward even if a fine set of features is considered. Also, advanced malware pack the original .dex file of applications and decrypt them at run-time by using a wrapper; therefore, they put a big challenge ahead of systems which rely heavily on features extracted before runtime.

Similar to identifier renaming detection, we use the full AMD dataset to evaluate the accuracy of our string encryption detection module when new apps are fed into the system over time. Our module for string encryption detection could achieve an average accuracy of 81.41% as shown in Fig. 5.4b. It improves soon after observing a few samples and increases up to 87.4% at maximum.

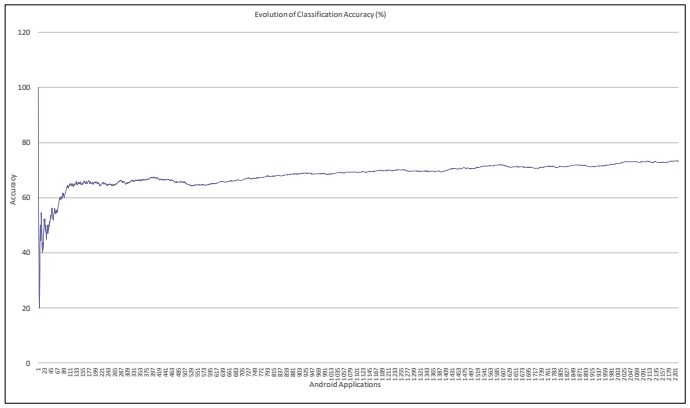
In total, 5499 samples were correctly classified (TP) as obfuscated, and 906 apps were mistakenly classified (FP) as obfuscated. In our studies, we found that malware samples make use of a wide range of cryptographic functions and encryption strategies which makes it challenging to consider a proper set of features in order to detect this particular type of obfuscation. A very simple way to do this is to simply extract some features from encrypted strings. Another advanced way is to extract features from encryption/decryption functions which are not always easily extractable as they are sometimes hidden in resource directory and are dynamically exercised at run-time.



(a) Identifier renaming detection



(b) String encryption detection



(c) Control flow obfuscation detection

Figure 5.4: Evolution of detector modules' accuracies over time.

Table 5.6: Performance metrics for each detection module.

Detector	TPR (Recall)	FPR (Inverse Recall)	Precision	F1 Score
Identifier Renaming	0.91	0.02	0.95	0.92
String Encryption	0.80	0.08	0.78	0.79
Control Flow Obfuscation	0.66	0.1	0.7	0.67

5.4.4 Control Flow Obfuscation Detection

Due to the limited number of samples we had for this type of obfuscation, we assess the average accuracy of our control flow obfuscation detection module over time only based on 80% of the applications collected here, and we keep the 20% remaining apps to test our system over unseen apps (recall Section 5.3.3) in the next sections (Section 5.4.6.3 and Section 5.4.6.4). Experimental results show that the corresponding ANDRODET module for control flow obfuscation detection is able to identify obfuscated apps with an average accuracy of 68.32% and a maximum accuracy of 73.4% on the final samples (Fig. 5.4c). This seems to be reasonable due to the limited number of samples we could feed into this module. Also, maximum accuracy percentage shows that this module would probably be able to have a better performance if it is fed with more training samples with a proper distribution of features.

Control flow obfuscation detection module could correctly label 898 samples as obfuscated (TP). Also, 429 samples were wrongly classified as obfuscated (FP). The main important reason for these relatively smaller values comparing with the ones achieved for identifier renaming and string encryption detection modules is the small amount of apps we had as ground truth for this type of obfuscation.

Generally speaking, accuracy plots for each of the obfuscation detection modules demonstrate the improvement of online learning algorithms over time when they observe more and more samples considering the fact that they do not need to be re-trained.

5.4.5 Performance Evaluation for Combined Techniques

To measure the performance of our system when apps are obfuscated using a combination of techniques, we extend the binary classification problem of each module to a multi-label classification problem and calculate the global accuracy using the same strategy we adopted for individual modules. Here, each detector module is tested and trained separately using the *EvaluatePrequential* class.

To achieve our goal and to be able to create a multi-label confusion matrix, we consider the presented encoding in Fig. 5.5. Thus, total number of combinations is 8 each of which is a binary representation of techniques used to obfuscate an application. For instance, 6 ('110') is a label which

shows that an app is obfuscated using both identifier renaming and string encryption techniques, and 0 ('000') demonstrates that the app is not obfuscated with any of these three techniques. However, we have excluded those labels for which we did not have any ground truth in our datasets.

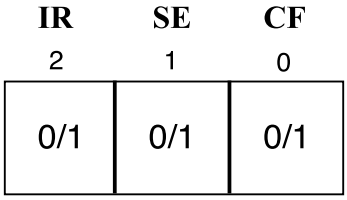


Figure 5.5: Multi-label encoding of obfuscation techniques.

Table 5.7: Confusion matrix for multi-label classification with MOA (real classes on rows and predicted classes on columns).

	N	CF	SE	IR	IR+SE
N	10,313	0	715	368	719
CF	210	758	0	0	142
SE	392	0	5,784	242	701
IR	103	0	99	5,513	277
IR+SE	309	0	300	213	1,224

N: No Obfuscation,
CF: Control Flow Obfuscation. SE: String Encryption,
IR: Identifier Renaming,

As it is clear from the confusion matrix (Table 5.7), the performance of each module obtained by dividing the true positive by false negative for that obfuscation technique is close to the values we separately evaluate on the previous sections. Also, the global accuracy of ANDRODET is approximately 80.66% considering the fact that some apps could be obfuscated with more than one technique. The prediction accuracy for apps which are obfuscated by identifier renaming and string encryption at the same time is 76.68% which stems in the fact that we had limited samples obfuscated with both techniques as ground truth.

5.4.6 Comparison Against Batch Learning Algorithms

This section compares the accuracy of our system to detect each type of obfuscation with a similar system based on batch learning algorithms over

unseen applications. To do so, we make use of a new dataset, known as PraGuard (recall Section 5.3.3). Also, we present and discuss the performance of both systems when a combination of techniques are used to obfuscate Android apps. Table 5.8 summarizes the results.

5.4.6.1 Identifier Renaming Detection

To compare the performance of ANDRODET's identifier renaming detection module with a similar system based on batch learning algorithms, we do the following experiment. We first feed our online learning module with a combined dataset of apps from AMD and PraGuard to measure its average accuracy using MOA. Then, we train another module based on batch learning algorithms with AMD to test it later over the PraGuard dataset using ATM tool.

Our results show that the online learning module improves its accuracy to 95.1% by observing further samples from PraGuard dataset. One the other hand, the module based on batch learning could achieve an accuracy of 91.5% (Table 5.8). The results obtained here highlights the adaptability power of online learning systems versus batch learning ones when new samples appear over time.

Table 5.8: Comparison of the accuracy between two systems for Android obfuscation detection based on online and batch learning algorithms (maximum accuracies).

Identifier Renaming		**String Encryption**		**Control Flow Obfuscation**	
MOA	ATM	MOA	ATM	MOA	ATM
95.1%	91.5%	85.6%	81.2%	73.7%	87.9%

5.4.6.2 String Encryption Detection

We adopt a Similar strategy to compare the performance of our online learning based module with another module which makes use of batch learning for string encryption detection on unseen applications, i.e., we observe how the accuracy of our learning module evolves over time when the new dataset (PraGuard) is fed into the system. We then train the batch learning based module with the AMD dataset and test it over PraGuard dataset to compare their accuracies.

Results confirm that the online module is able to update its model incrementally by observing new samples, and, thus, could reach an accuracy of 85.6% compared to the batch learning module with a lower accuracy. Although the difference is not big, this result bolds the advantage of online learning algorithms over batch learning ones in improving their built model without the need of time consuming training procedure.

5.4.6.3 Control Flow Obfuscation Detection

Due to the limited available ground truth for control flow obfuscated apps, 80% (2220 apps) of the apps (1,387 obfuscated apps from AMD and 1,387 non-obfuscated apps from F-Droid) is used to evaluate our online learning module (as performed in Sec. 5.4.4), and 20% (554 apps) is used to inspect how our system's accuracy evolves when new apps appear, and, also, to compare its performance with a similar module based on batch learning algorithms.

The accuracies obtained from both systems show that the batch learning based module can predict the label of unseen apps with a higher accuracy. However, there is a major difference between our test samples used for this module with the other two modules (Sec. 5.4.6.1 and 5.4.6.2). The difference is that unseen apps are fed into the system from the same datasets (AMD and F-Droid) which were used for evaluating our online module, and, thus, are expected to have similar features. In other words, unseen apps do not add much information to the previously built model of our module.

5.4.6.4 Combined Obfuscation Techniques

In a final assessment, we repeat the same experiment as we did in Sec. 5.4.5, but on unseen applications. Thus, we use the PraGuard dataset as ground truth for identifier renaming and string encryption techniques, and the remaining 20% of apps from AMD and F-Droid as ground truth for control flow obfuscation. We compare our results with another system based on batch learning algorithms. For ANDRODET, we inspect how our system can extend its built model when new apps are fed into the system and when they might use a variety of obfuscation techniques.

As it is clear from the confusion matrices of the two detection systems (Table 5.9 and Table 5.10), the global accuracy of ANDRODET when it is

fed with more unseen applications and is tested at the same time is around 83.34% which shows a minor improvement comparing with the one obtained in Section 5.4.5. On the contrary, the global accuracy of a similar system based on batch learning algorithms is around 85.64%. Also, accuracies of detector modules which can be obtained from these matrices are aligned with the results we achieved before (Table 5.8).

Table 5.9: Confusion matrix for multi-label classification with MOA on unseen applications (real classes on rows and predicted classes on columns).

	N	**CF**	**SE**	**IR**	**IR+SE**
N	11,913	0	715	374	887
CF	145	1,018	0	0	224
SE	459	0	7,196	368	591
IR	97	0	166	7,049	175
IR+SE	229	0	216	267	2,829

N: No Obfuscation,
IR: Identifier Renaming,
SE: String Encryption,
CF: Control Flow Obfuscation.

In particular, the individual accuracy of the control flow obfuscation detection module on unseen applications using batch learning algorithms is slightly higher than the accuracy of the same module based on online learning algorithms. This is vice versa for the other two obfuscation techniques, namely identifier renaming and string encryption, i.e. the accuracies of detector modules which make use of online learning algorithms are higher than the same modules which are based on batch learning algorithms. Also, the system which works based on batch learning algorithms outperforms ANDRODET when it comes to apps that are obfuscated by both identifier renaming and string encryption techniques.

5.4.7 Performance Comparison: Time and Memory

One key advantage of using online learning algorithms in classification is their ability to update their model upon observing new samples opposite to batch learning algorithms which do need to be re-trained after specific intervals in order to preserve their accuracies over time. Re-training process needs a considerable amount of memory as well. Thus, to compare

Table 5.10: Confusion matrix for multi-label classification with ATM on unseen applications (real classes on rows and predicted classes on columns).

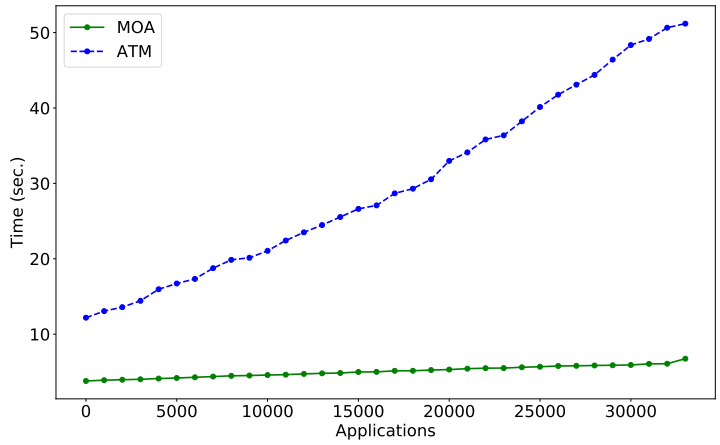
	N	**CF**	**SE**	**IR**	**IR+SE**
N	12,021	0	709	369	790
CF	45	1,216	0	0	126
SE	459	0	7,146	368	641
IR	92	0	149	6,877	369
IR+SE	254	0	216	267	2,804

N: No Obfuscation,
IR: Identifier Renaming,
SE: String Encryption,
CF: Control Flow Obfuscation.

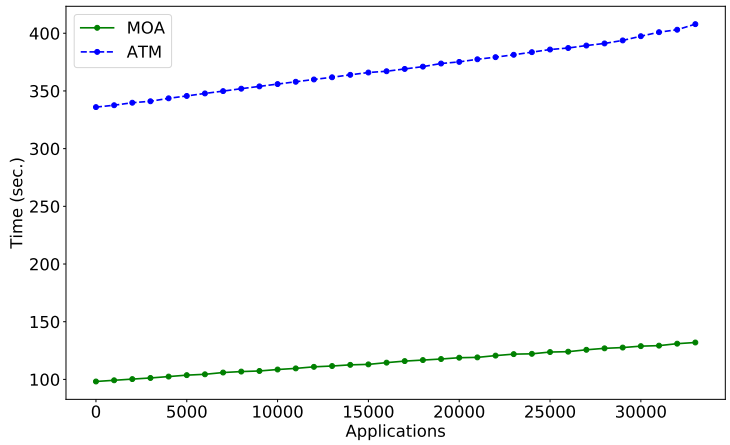
ANDRODET with a similar system based on batch learning algorithms (the system discussed in Section 5.4.6.4) in terms of time and memory we conduct the following experiment.

For time analysis, we assume that the batch learning system needs to be re-trained after classifying every 1000 samples (1000 epochs). With this assumption, we start classifying the whole applications (recall Table 5.1); but, here, the system is re-trained after classifying every 1000 samples. Thus, the time for each epoch is calculated by summing up the time which is needed to train, and, then, test the system over next 1000 samples. And, the final cumulative time is the sum of time spent in all epochs until it classifies all applications. For ANDRODET, each epoch's time is obtained by only measuring the time which is used for classification. We analyze memory usage based on the same assumption as shown Fig. 5.6. Here, we exclude the amount of time and memory which is used for hyper-parameter tuning in both systems. However, we consider the time which is needed to train both systems at the beginning.

As it is clear, ANDRODET outperforms a similar system based on batch learning algorithms in both time and memory consumption on a medium size dataset. If the dataset size increases time by time, and if the built model is needed to be updated in shorter intervals, this difference will most probably be higher between online learning systems and batch learning ones. Another important aspect is to inspect the amount of memory which is consumed as the dataset grows in size over time. Based on our observations, ANDRODET consumed 33.79 MB at maximum as the dataset increased



(a) Computation Time.



(b) Memory Consumption.

Figure 5.6: Comparison of time and memory consumption between online learning algorithms using MOA (a) and batch learning algorithms using ATM (b) for Android obfuscation detection.

to around 34K apps. In contrary, the system based on batch learning algorithms consumed 71.89 MB of RAM memory as more samples were added to the training set over time.

5.5 Threats to Validity

This section discusses a number of potential limitations we encountered in our work. Our datasets contain two main issues that could impact the

validity of our results. On the one hand, they do not contain a uniform distribution for all combinations of obfuscation techniques. For example, there is not a sample in our datasets in which string encryption and control flow obfuscation have been jointly applied. To the best of our knowledge, there is no dataset that contains such a type of application. Therefore, the analysis on the effectiveness of this approach for these types is left for future work. On the other hand, our datasets contain apps which are control flow obfuscated using a single tool (i.e., Allatori). As a consequence, apps which are obfuscated with other tools may evade detection by ANDRODET if the techniques they employ are quite different.

State-of-the-art Android reverse engineering tools are shown not to work properly in all cases. Thus, systems that make use of features extracted by these tools are prone to errors. For instance, disassemblers may make mistakes which could in turn hide information to the systems that use the result of disassembly. Also, tools which extract control flow graphs are not perfect, especially when apps adopt advanced anti-analysis techniques.

Advanced code obfuscation techniques in Android may use a combination of transformations [238]. Although ANDRODET is modular and can detect if a malware is obfuscated using more than one technique, it does not consider all possible combinations which might exist in the wild. However, there is not a comprehensive and systematic study to report the prevalence of adopting various combinations of Android obfuscation techniques at the moment. Moreover, advanced malware specimens use a wide range of techniques to evade malware analysis systems which can affect our system.

5.6 Related Work

Many prior works have attempted to address the problem of handling obfuscation in Android. On the one hand, the goal of several works is to carry out a process without any impact despite of obfuscation. Particularly, a matter of interest is malware analysis. In this regard, [239] propose RevealDroid, a system for malware detection and family identification in an obfuscation-resilient manner. On the other hand, Zhang *et al.* aim to detect repackaged applications by inspecting the user interactions in the graphical interface [240]. The same problem is addressed by CodeMatch, which is able to deal with other types of obfuscation such as code slicing [241].

The works described so far consider obfuscation as an obstacle to be saved to achieve a goal of a different nature. In this work, the detection of obfuscation is indeed the target of the approach. In this regard, two actions have been considered in other works, either detecting obfuscation or even attempting to deobfuscate the app. Each one is described in the following.

With respect to obfuscation detection, in 2018 Dong *et al.* have carried out a large-scale investigation. They focus on four types of obfuscation, namely identifier renaming, string encryption, Java reflection and packing. For each of them, they propose a lightweight detector that leverages signatures and machine learning techniques. Their approach is assessed using a dataset formed by 114,560 apps from both goodware and malware. To detect identifier renaming and string encryption, they use Support Vector Machine (SVM) as technique and 3-grams as features. To date, their work is the most similar to ours. In a similar vein, Wang and Rountev attempted to detect the tool that has been applied. For this purpose, they take 282 apps from F-Droid and obfuscate them using different tools using several configurations. These configurations indicate the type of obfuscation applied. Interestingly, these configurations involve identifier renaming, string encryption, package modification and control flow obfuscation. Using 10 sets of strings (e.g., method names, package names, etc.), their approach also relies upon SVMs [20]. In their approach, they reach 97.5 % of accuracy for obfuscator detector, and similar rates when it comes to detect which configuration has been applied in each tool. As compared to this work, their dataset is significantly smaller. Moreover, they do not deal with the re-training aspect.

Concerning deobfuscation attempts, [242] presents early results on deobfuscation against ProGuard tool. Their approach is based on comparing the similarity of some portions of the code against a database filled up with unobfuscated code. On the other hand, Yoo *et al.* propose a string deobfuscation technique to improve malware detection ratios [243]. This technique is based on running the app, intercepting all results coming from functions returning strings, and, then, repackaging the app replacing the original strings with these intercepted results. In this way, no matter what kind of encryption is applied, the tool is able to get the decrypted value. Their method outperforms other tool-specific mechanisms such as dex-oracle[2]. Another deobfuscation work is presented by Bischel *et al.* [244].

[2]https://github.com/CalebFenton/dex-oracle , last accessed March 2018

Their focus is on identifier renaming obfuscation, and their approach bases on comparing a given identifier with a large database of non-obfuscated ones. As compared to these attempts, our proposal does not aim to deobfuscate, but can serve as starting point to address this in future. In particular, the output of ANDRODET is useful to spot the type of obfuscation at stake, which can be considered to apply focused deobfuscation techniques. Moreover, our approach considers several types of obfuscation.

5.7 Conclusion

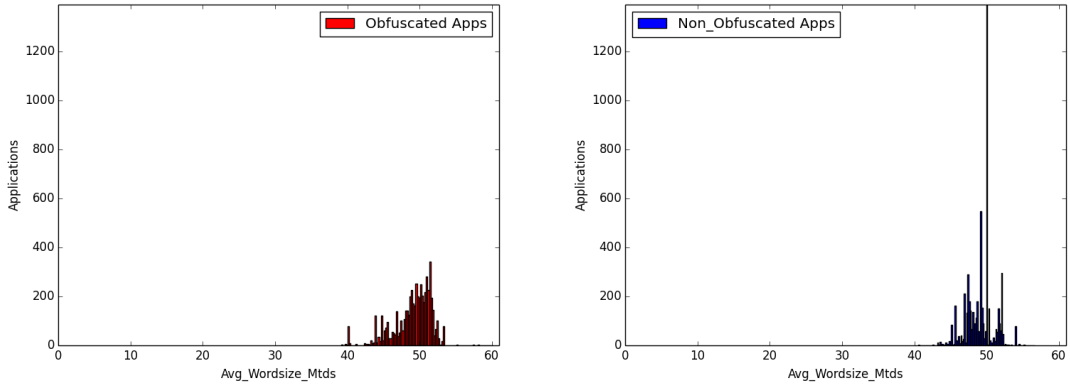
Obfuscation is one of the main obstacles when it comes to Android app analysis. Thus, having a mechanism to detect the type of existing obfuscation (if any) can contribute saving resources for analysis. Indeed, particular analysis techniques may be applied once this detection has been done. To contribute in this direction, in this work ANDRODET has been proposed. ANDRODET shows promising accuracy ratios for detecting identifier renaming, string encryption, and control flow obfuscation. Moreover, it requires moderate training needs and can be configured to work in online basis, that is, with incremental training. To foster further research in this area, both ANDRODET sources and the experimental dataset are freely available.

Several issues are devised as future research directions. First, addressing other types of obfuscation. Second, refining the feature set to improve the current accuracy of modules. Last but not least, extracting features by directly parsing the header of Dex files which will save more time and will compensate the limitations of Android reverse engineering tools.

5.8 Supplemental Data

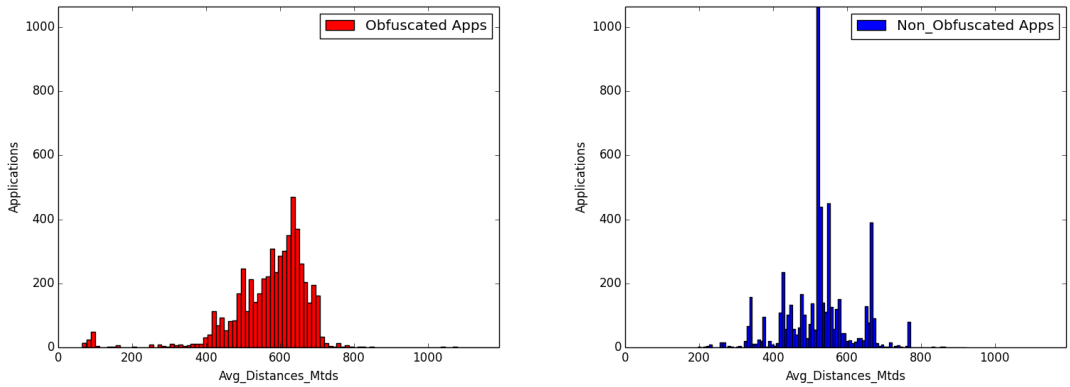
5.8.1 Distribution of Features for Identifier Renaming Detection

This section presents the distribution of attributes in the methods and classes which were extracted from obfuscated and non-obfuscated samples of AMD dataset.



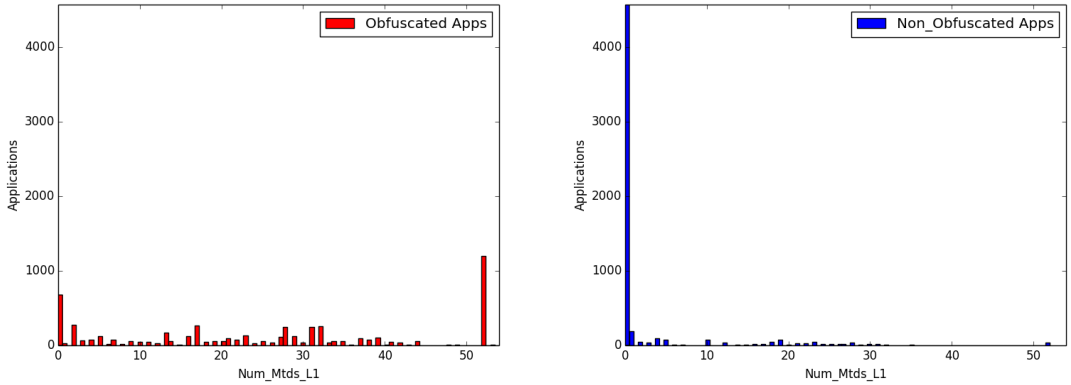
(a) Obfuscated Applications. **(b)** Non-Obfuscated Applications.

Figure 5.7: Distribution of the average wordsize of methods in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. **(b)** Non-Obfuscated Applications.

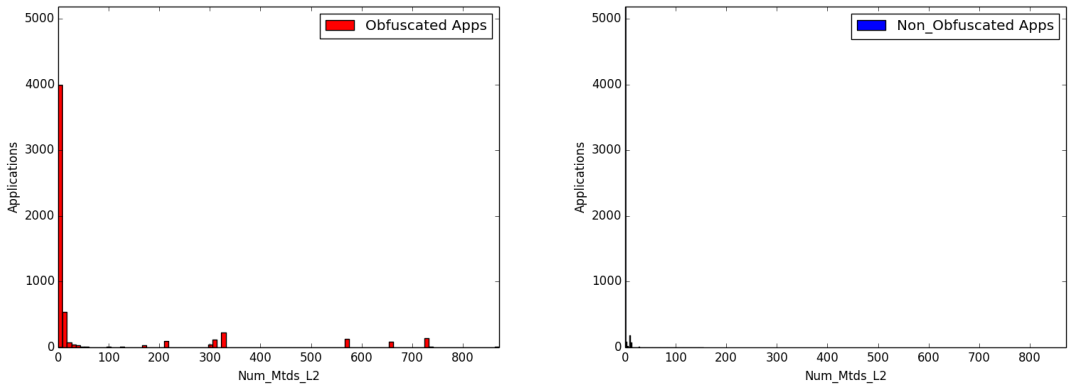
Figure 5.8: Distribution of the average ASCII distances between consecutive extracted methods in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications.

(b) Non-Obfuscated Applications.

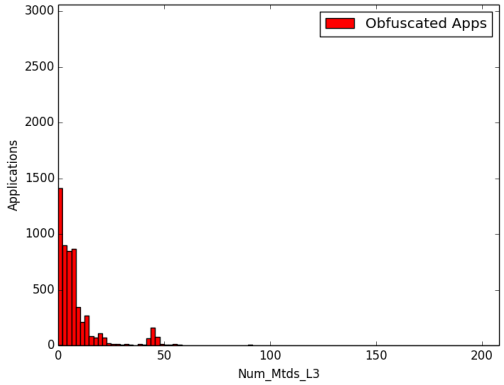
Figure 5.9: Distribution of methods with length 1 in (a) obfuscated and (b) non-obfuscated apps.



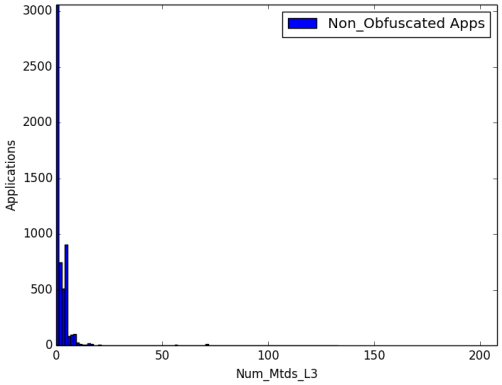
(a) Obfuscated Applications.

(b) Non-Obfuscated Applications.

Figure 5.10: Distribution of methods with length 2 in (a) obfuscated and (b) non-obfuscated apps.

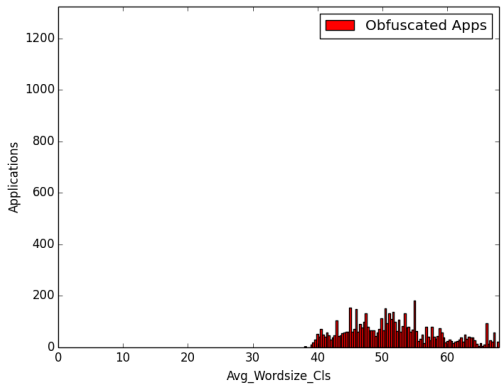


(a) Obfuscated Applications.

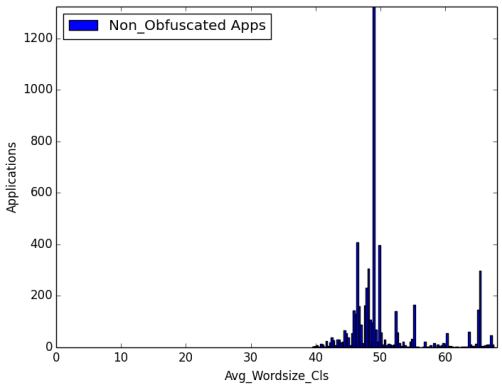


(b) Non-Obfuscated Applications.

Figure 5.11: Distribution of methods with length 3 in (a) obfuscated and (b) non-obfuscated apps.

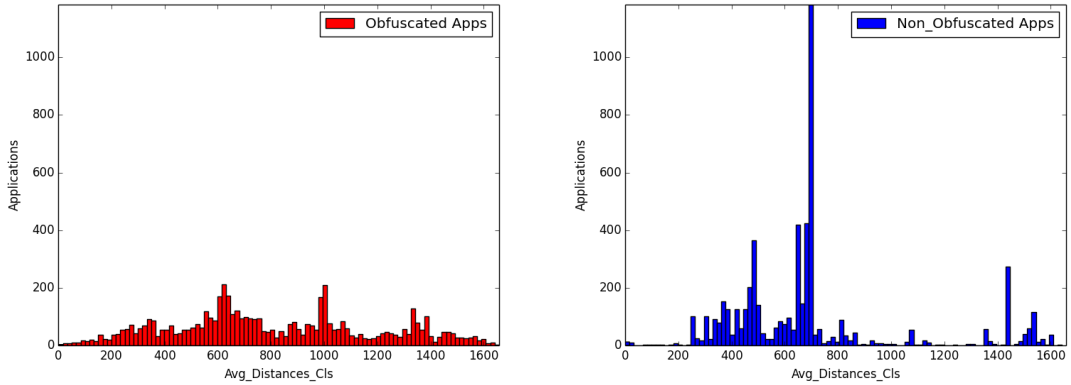


(a) Obfuscated Applications.



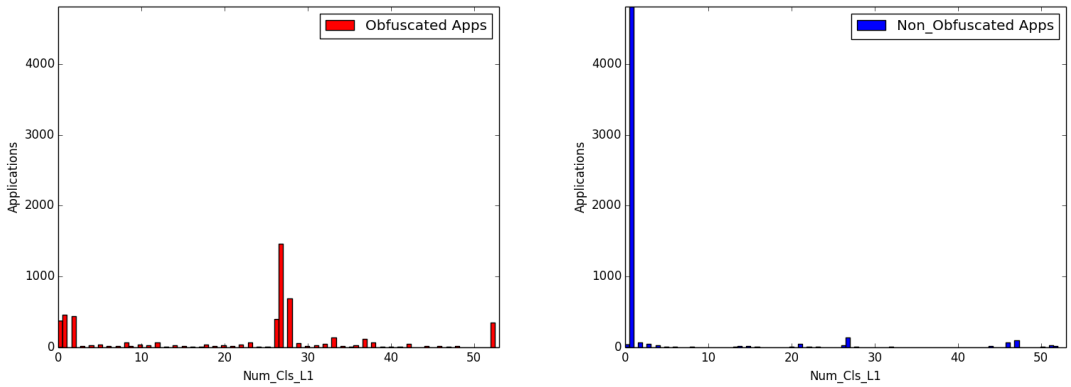
(b) Non-Obfuscated Applications.

Figure 5.12: Distribution of the average wordsize of classes in (a) obfuscated and (b) non-obfuscated apps.



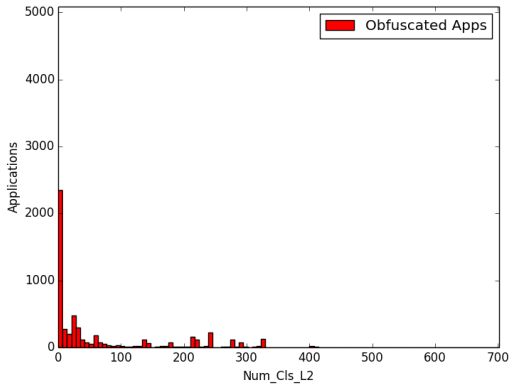
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.13: Distribution of the average ASCII distances between consecutive extracted classes in (a) obfuscated and (b) non-obfuscated apps.

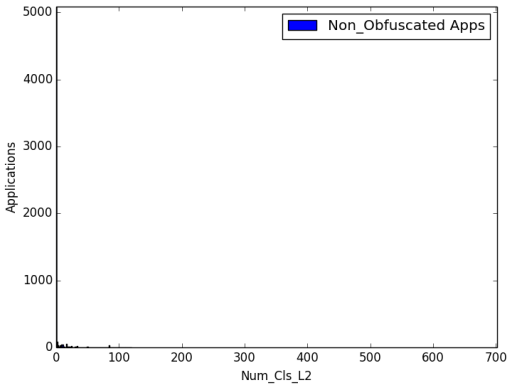


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.14: Distribution of classes with length 1 in (a) obfuscated and (b) non-obfuscated apps.

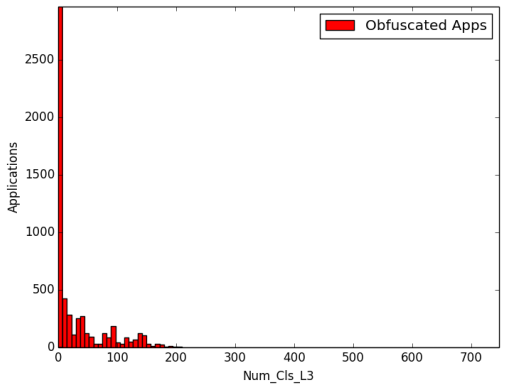


(a) Obfuscated Applications.

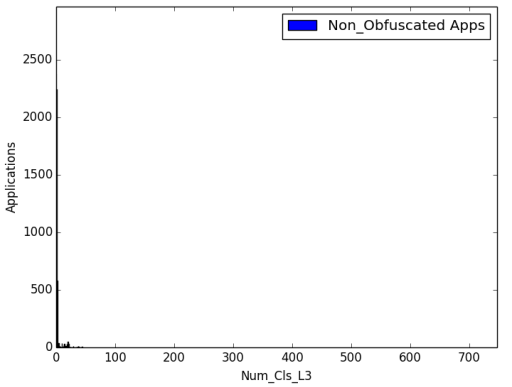


(b) Non-Obfuscated Applications.

Figure 5.15: Distribution of classes with length 2 in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications.

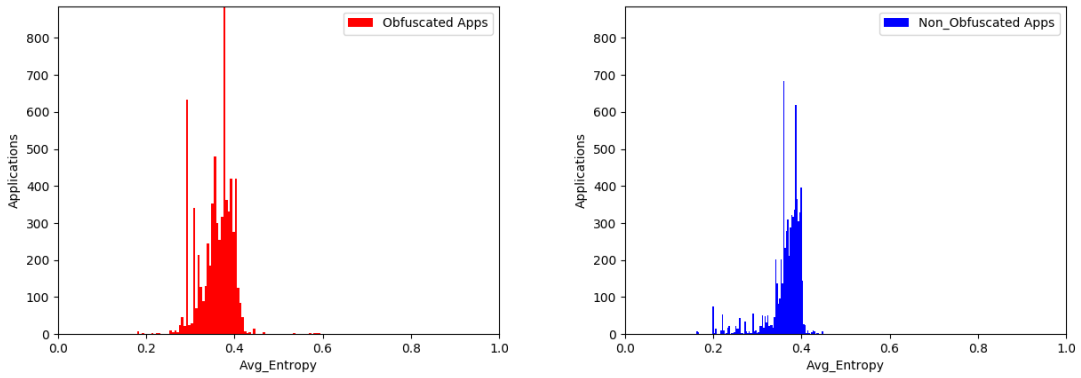


(b) Non-Obfuscated Applications.

Figure 5.16: Distribution of classes with length 3 in (a) obfuscated and (b) non-obfuscated apps.

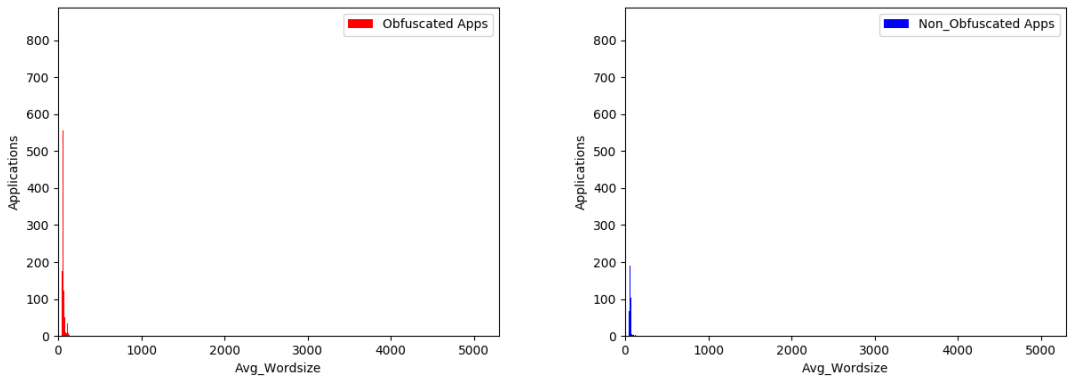
5.8.2 Distribution of Features for String Encryption Detection

This section presents the distribution of attributes in the strings which were extracted from obfuscated and non-obfuscated samples of AMD dataset.



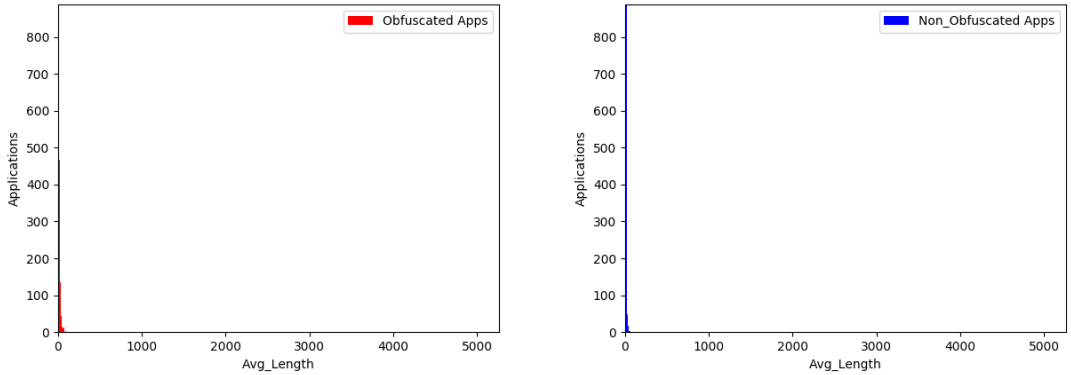
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.17: Distribution of the average entropy of strings in (a) obfuscated and (b) non-obfuscated apps.



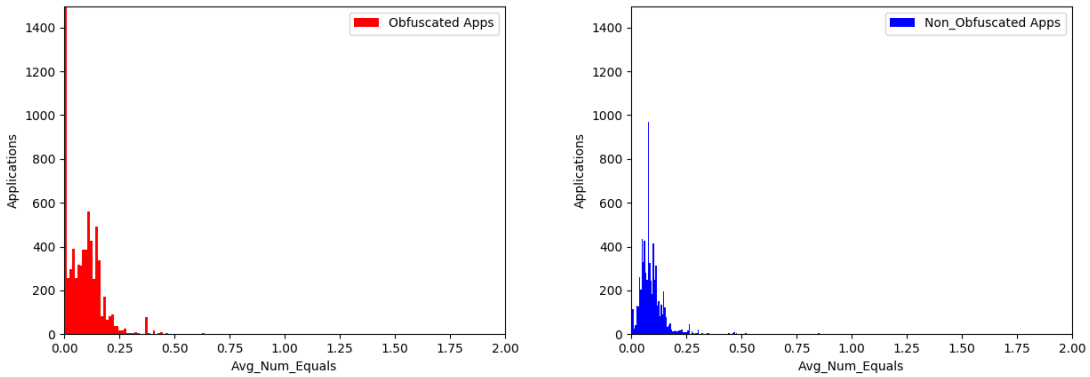
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.18: Distribution of the average wordsize of strings in (a) obfuscated and (b) non-obfuscated apps.



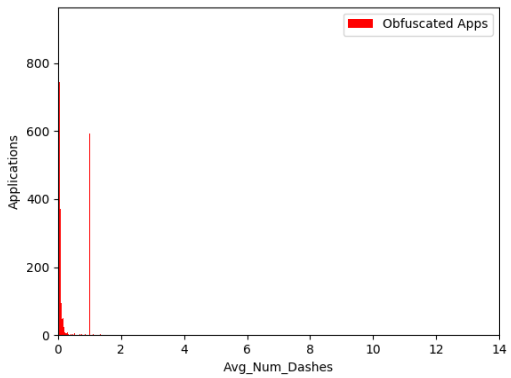
(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.19: Distribution of the average length of strings in (a) obfuscated and (b) non-obfuscated apps.

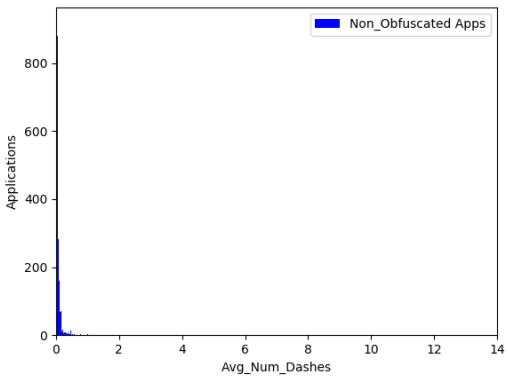


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.20: Distribution of the average number of '=' characters in (a) obfuscated and (b) non-obfuscated apps.

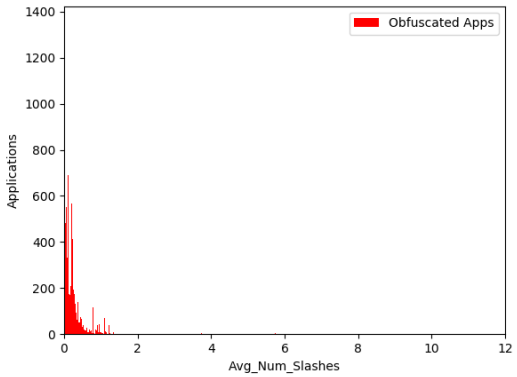


(a) Obfuscated Applications.

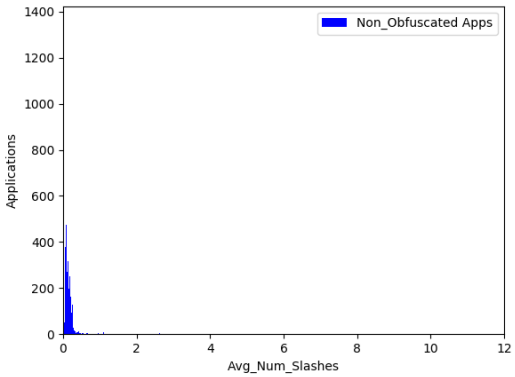


(b) Non-Obfuscated Applications.

Figure 5.21: Distribution of the average number of '-' characters in (a) obfuscated and (b) non-obfuscated apps.

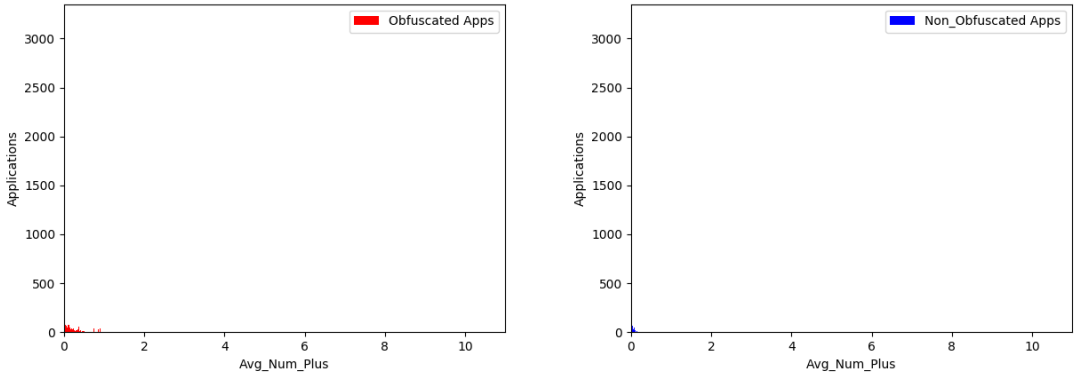


(a) Obfuscated Applications.



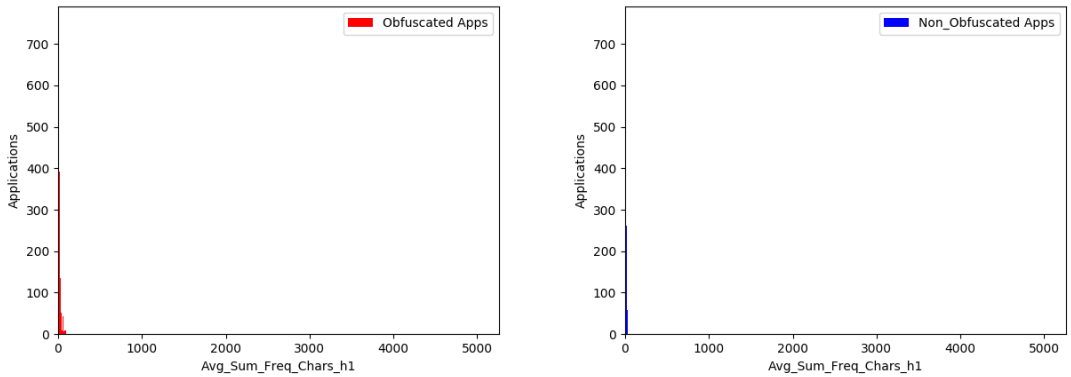
(b) Non-Obfuscated Applications.

Figure 5.22: Distribution of the average number of '/' characters in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.23: Distribution of the average number of '+' characters in (a) obfuscated and (b) non-obfuscated apps.

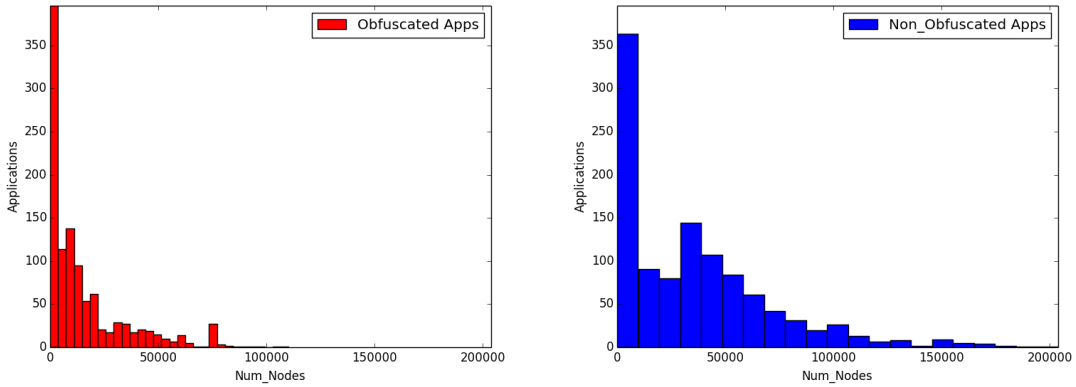


(a) Obfuscated Applications. (b) Non-Obfuscated Applications.

Figure 5.24: Distribution of the average sum of repetitive characters in (a) obfuscated and (b) non-obfuscated apps.

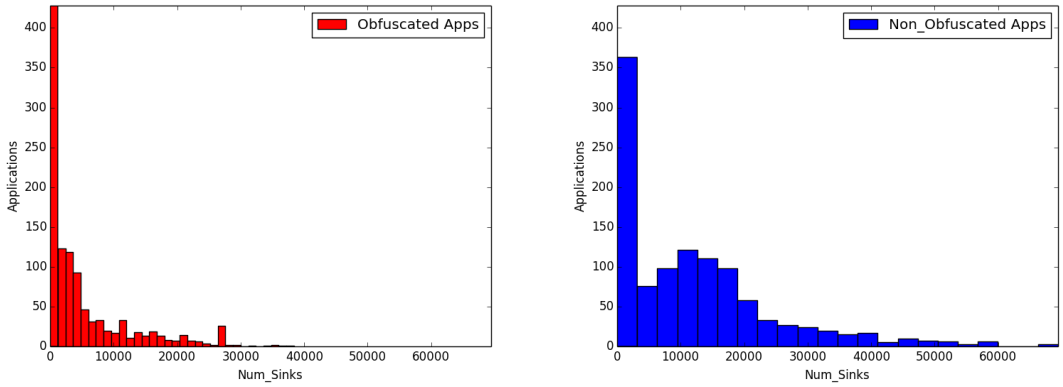
5.8.3 Distribution of Features for Control Flow Obfuscation Detection

This section presents the distribution of attributes extracted from the CFG and Dalvik bytecode of obfuscated (from AMD dataset) and non-obfuscated (from F-Droid dataset) samples.



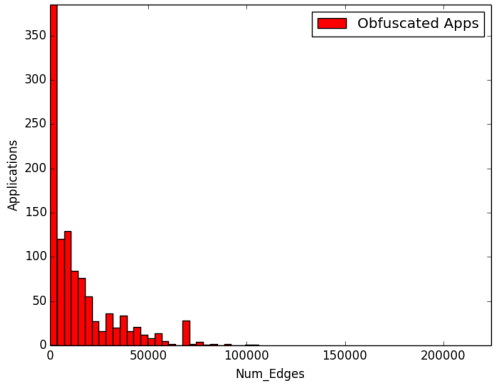
(a) Obfuscated Applications. **(b)** Non-Obfuscated Applications.

Figure 5.25: Distribution of the number of nodes in the CFG of (a) obfuscated and (b) non-obfuscated apps.

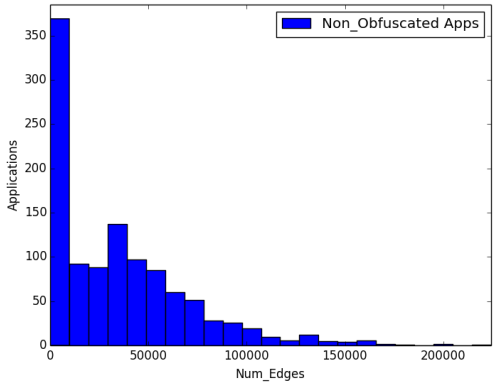


(a) Obfuscated Applications. **(b)** Non-Obfuscated Applications.

Figure 5.26: Distribution of the number of sinks in the CFG of (a) obfuscated and (b) non-obfuscated apps.

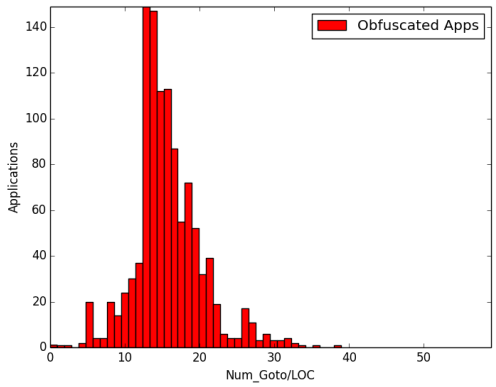


(a) Obfuscated Applications.

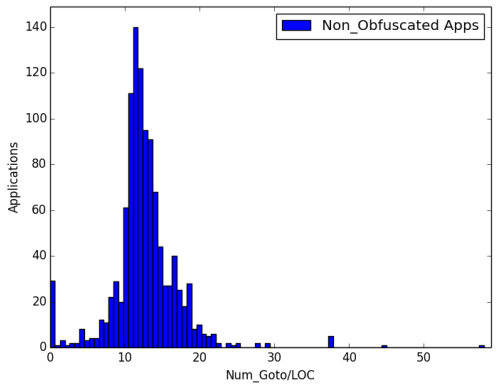


(b) Non-Obfuscated Applications.

Figure 5.27: Distribution of the number of edges in the CFG of (a) obfuscated and (b) non-obfuscated apps.

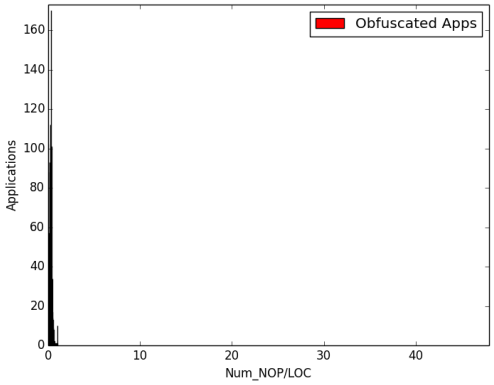


(a) Obfuscated Applications.

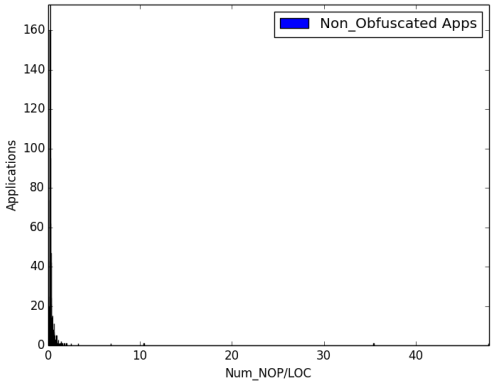


(b) Non-Obfuscated Applications.

Figure 5.28: Distribution of the number of Goto instructions per line of code in (a) obfuscated and (b) non-obfuscated apps.

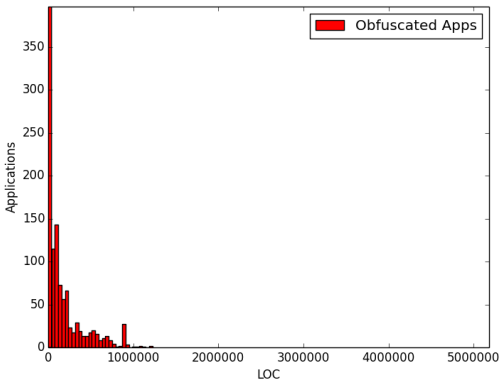


(a) Obfuscated Applications.

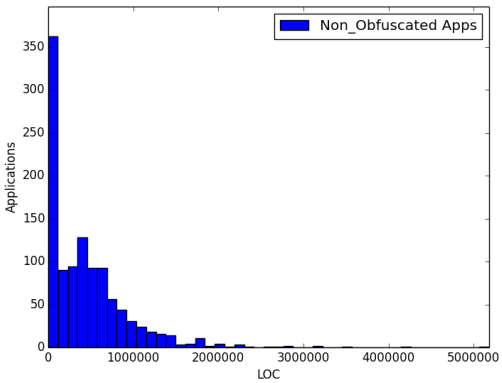


(b) Non-Obfuscated Applications.

Figure 5.29: Distribution of the number of NOP instructions per line of code in (a) obfuscated and (b) non-obfuscated apps.

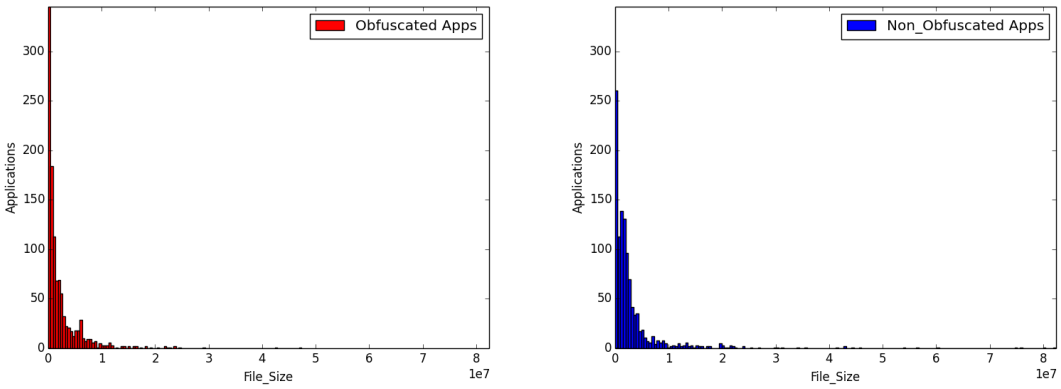


(a) Obfuscated Applications.



(b) Non-Obfuscated Applications.

Figure 5.30: Distribution of the total number of lines of code in (a) obfuscated and (b) non-obfuscated apps.



(a) Obfuscated Applications. **(b)** Non-Obfuscated Applications.

Figure 5.31: Distribution of the total number of lines of code in (a) obfuscated and (b) non-obfuscated apps.

6

Conclusions

The main focus of this dissertation is on Android malware triage due to the limitations of both static and dynamic analysis tools. We have designed and implemented a novel tool, called TriFlow, that automatically scores Android apps based on a forecast of their information flows and their associated risk. Our approach relies on a probabilistic model for information flows and a measure of how significant each flow is. Both items are experimentally obtained from a dataset containing benign and malicious apps. After this training phase, the models are used by a fast mechanism to triage apps; thus, providing a queuing discipline for the pool of apps waiting for a precise information flow analysis. Our experimental results suggest that TriFlow provides a sensible ordering based on the potential interest of the app. Given the huge amount of computational resources demanded by information flow analysis tools, we believe this could be very helpful to maximize the expected utility when dealing with large pools of apps. Additionally, TriFlow could be used as a standalone risk metric for Android apps, providing a complementary perspective to alternative risk assessment approaches based on permissions and other static features. Finally, to encourage further research in this area, we have made our results and implementation available online.

During our studies, we found that malware labels are not necessarily consistent with apps' behavior in different datasets due to the lack of appropriate standards to name malware samples. Thus, in our second contribution, we have discussed the problem of how family labels in Android malware relate to the behavior of their samples. Our approach relies on modeling apps through their information flows and then characterizing behavior by patterns of such flows. We have also conducted a cluster analysis to identify groups of apps whose patterns of information flows are similar. Our experimental results show that the notion of Android malware family

does not necessarily imply that all of its samples behave similarly, and also that different malware families sometimes behave identically.

Another barrier we encountered during our studies on Android malware triage was the vast application of anti-analysis techniques in general and, obfuscation in particular, to Android malware by both commercial and non-commercial tools which hindered their accurate analysis. Therefore, in our last contribution, we have developed and presented AndrODet, a tool for detecting three common obfuscation techniques in Android apps, including identifier renaming, string encryption, and control flow obfuscation. We believe this tool is one of the pioneers in detecting obfuscation in Android apps and is a big step forward to remove the barriers currently exist in Android malware analysis. AndrODet requires moderate training needs and can be configured to work in online basis, that is, with incremental training. To foster further research in this area, both AndrODet sources and the experimental dataset have been made freely available.

6.1 Awards

The work contained in this dissertation has resulted in two outstanding and competitive awards as follows:

- **Third Place Award**,
  From CSAW-Europe Best Applied Security Research Competition,
  For the paper, "TriFlow: Triaging Android Applications using Speculative Information Flows"
- **Best Previously Published Paper Award**,
  From 4th Spanish National Cybersecurity Research Conference (JNIC),
  For the paper, "A Summary of TriFlow: Triaging Android Applications using Speculative Information Flows"

6.2 Tools

Two well-documented tools have also been released for public use in order to foster research in related areas. Below, we provide the readers with a summary of their functionalities and their access links:

- **TriFlow**:
  This tool builds up an efficient probabilistic model to predict the occurrence of information flows in Android apps. It also weight flows based on their prevalence in Android malware and benign apps. Then, it triages new apps based on a forecast of flows that may appear in the apps and their amount of maliciousness. The tool is continuously watched and updated for better performance upon researchers' feedback.,
  Download Link: https://github.com/OMirzaei/TriFlow
- **AndrODet**:
  This tool is an adaptive modular system to detect three common types of obfuscation in Android apps, including identifier renaming, string encryption and control flow obfuscation. A separate module has been considered to detect each type of obfuscation by extracting relevant features. Each module improves its accuracy over time by observing new samples and learning from them on the fly without the need to be re-trained. AndrODet can be easily modified to detect more types of obfuscation techniques.,
  Download Link: https://github.com/OMirzaei/AndrODet

6.3 Research Visits

The following universities were also visited during this PhD educational period:

- **University College London (UCL)**:
  During my PhD, I have visited Dr. Gianluca Stringhini at the Information Security Research Group from August to November 2017. Since then, we have started a collaboration which is still ongoing and is aimed to lead to a top publication.

6.4 Future Work

Though Android malware has been studied for some years and several contributions are made to improve the security of smartphones running Android OS, this area of research is not yet mature enough as we still see

attack reports on a daily basis. Based on the results and feedback we obtained from this thesis, we highlight and present some areas that can be explored as future works.

Obfuscation. Based on our observations, code obfuscation is one of the most prevalent anti-analysis techniques used by malware to evade analysis. Moreover, some malware use runtime-based obfuscation which makes their inspection much harder. Though quite common, there is not a comprehensive study of obfuscation on big amount of apps in the wild, neither an accurate tool to deobfuscate apps in order to facilitate their precise analysis. Although recent works have tried to address both runtime-based and regular types of obfuscation such as identifier renaming and string encryption, their tools do have some shortcomings. We believe there is much more room here to investigate different anti-analysis techniques which are used by Android malware at the first step. Also, much more is needed to be done to develop efficient obfuscation detection tools.

Packing. Packing is another common method used by malware to evade static analysis tools. Recent works have studied this problem precisely and have publicly released some tools to detect and unpack Android apps which are packed by different techniques. However, these tools have shown to be not efficient for all virtual machines and all versions of Android OS. Also, few of them have not been well supported and documented as well. Thus, we believe more research works can be conducted in this area too.

Ransomware. Security threat reports of most well-known cybersecurity companies show that Android ransomware has increased in number once again after its emergence in 2014. Android screen lockers were the dominant types of malware in 2017 and other variants of ransomware are appearing again in 2018. Based on these observations, we believe another open area of research is to explore different types of Android ransomware and to develop tools and countermeasures to defeat all types of threats which may arise from this type of malware.

Mining Cryptocurrencies. By the appearance of different cryptocurrencies, including Bitcoin, Monero and Zcash, mining cryptocurrencies is an appealing target for attackers. Although Google has recently banned on-device mining on Android hardware, it still allows mining as long as the processing takes place in the cloud. Furthermore, even though mining requires a huge amount of processing power and vast networks of devices,

significant worth of cryptocurrencies is a good motivator for attackers to devise mining techniques on Android devices. We believe research community should begin exploring the usage of cryptocurrency mining in Android malware and develop appropriate tools to block crytocurrency mining on smartphones running Android operating system.

References

[1] Nikolay Elenkov. *Android security internals: An in-depth guide to Android's security architecture*. No Starch Press, 2014.

[2] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *NDSS*, 2014.

[3] Number of smartphone users worldwide from 2014 to 2020 (in billions). https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/, (accessed July 8, 2018).

[4] Android market share in july 2018. http://gs.statcounter.com/os-market-share/mobile/worldwide/#monthly-201807-201807-bar, (accessed July 8, 2018).

[5] Google play. https://play.google.com/store?hl=en, (accessed July 8, 2018).

[6] Google play protect. https://www.android.com/play-protect/, (accessed July 8, 2018).

[7] Shuaike Dong, Menghao Li, Wenrui Diao, Xiangyu Liu, Jian Liu, Zhou Li, Fenghao Xu, Kai Chen, Xiaofeng Wang, and Kehuan Zhang. Understanding android obfuscation techniques: A large-scale investigation in the wild. *arXiv preprint arXiv:1801.01633*, 2018.

[8] Yue Duan, Mu Zhang, Abhishek Vasisht Bhaskar, Heng Yin, Xiaorui Pan, Tongxin Li, Xueqiang Wang, and X Wang. Things you may not know about android (un) packers: a systematic study based on whole-system emulation. In *25th Annual Network and Distributed System Security Symposium, NDSS*, pages 18–21, 2018.

[9] Kimberly Tam, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh, and Lorenzo Cavallaro. The evolution of android malware and android

analysis techniques. *ACM Computing Surveys (CSUR)*, 49(4):76, 2017.

[10] Mobile threat report. Technical report, McAfee, 2018.

[11] Symantec internet security threat report trends for 2017. Technical report, Symantec Corporation, 2018.

[12] Jingjing Ren, Martina Lindorfer, Daniel J. Dubois, Ashwin Rao, David Choffnes, and Narseo Vallina-Rodriguez. Bug fixes, improvements, ... and privacy leaks. In *25th Annual Network and Distributed System Security Symposium, NDSS*, volume 2018, 2018.

[13] Abbas Razaghpanah, Rishab Nithyanand, Narseo Vallina-Rodriguez, Srikanth Sundaresan, Mark Allman, and Christian Kreibich Phillipa Gill. Apps, trackers, privacy, and regulators. In *25th Annual Network and Distributed System Security Symposium, NDSS*, volume 2018, 2018.

[14] Elleen Pan, Jingjing Ren, Martina Lindorfer, Christo Wilson, and David Choffnes. Panoptispy: Characterizing audio and video exfiltration from android applications. *Proceedings on Privacy Enhancing Technologies*, 2018, 2018.

[15] Marcos Sebastián, Richard Rivera, Platon Kotzias, and Juan Caballero. Avclass: A tool for massive malware labeling. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 230–253. Springer, 2016.

[16] Médéric Hurier, Guillermo Suarez-Tangil, Santanu Kumar Dash, Tegawendé F Bissyandé, Yves Le Traon, Jacques Klein, and Lorenzo Cavallaro. Euphony: Harmonious unification of cacophonous anti-virus vendor labels for android malware. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 425–435. IEEE Press, 2017.

[17] Guillermo Suarez-Tangil, Juan E Tapiador, Pedro Peris-Lopez, and Jorge Blasco. Dendroid: A text mining approach to analyzing and classifying code structures in android malware families. *Expert Systems with Applications*, 41(4):1104–1117, 2014.

[18] Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial perturbations against deep neural networks for malware classification. *arXiv preprint arXiv:1606.04435*, 2016.

[19] Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, Francesco Mercaldo, and Corrado Aaron Visaggio. Impact of code obfuscation on android malware detection based on static and dynamic analysis. In *4th International Conference on Information Systems Security and Privacy*, pages 379–385. Scitepress, 2018.

[20] Yan Wang and Atanas Rountev. Who changed you?: obfuscator identification for android. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems*, pages 154–164. IEEE Press, 2017.

[21] Mario Linares-Vásquez, Gabriele Bavota, and Camilo Escobar-Velásquez. An empirical study on android-related vulnerabilities. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 2–13. IEEE, 2017.

[22] Karim Yaghmour. *Embedded Android: Porting, Extending, and Customizing*. " O'Reilly Media, Inc.", 2013.

[23] Lok Kwong Yan and Heng Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 569–584, 2012.

[24] Anthony Desnos and Geoffroy Gueguen. Android: From reversing to decompilation. *Proc. of Black Hat Abu Dhabi*, pages 77–101, 2011.

[25] Huasong Meng, Vrizlynn LL Thing, Yao Cheng, Zhongmin Dai, and Li Zhang. A survey of android exploits in the wild. *Computers & Security*, 2018.

[26] Dexdump. googlesource.com/platform/dalvik/+/eclairrelease/dexdump/DexDump.c, (accessed February 10, 2018).

[27] Dex2jar. https://bitbucket.org/pxb1988/dex2jar, (accessed February 10, 2018).

[28] Androguard. github.com/androguard/androguard, (accessed February 10, 2018).

[29] Apktool. https://ibotpeaches.github.io/Apktool, (accessed February 10, 2018).

[30] Rowland Yu. Android packers: facing the challenges, building solutions. In *Proceedings of the 24th Virus Bulletin International Conference*, 2014.

[31] Bodong Li, Yuanyuan Zhang, Juanru Li, Wenbo Yang, and Dawu Gu. Appspear: Automating the hidden-code extraction and reassembling of packed android malware. *Journal of Systems and Software*, 2018.

[32] Kanae Yoshida, Hironori Imai, Nana Serizawa, Tatsuya Mori, and Akira Kanaoka. Understanding the origins of weak cryptographic algorithms used for signing android apps. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, pages 713–718. IEEE, 2018.

[33] Guillermo Suarez-Tangil, Juan E. Tapiador, Pedro Peris, and Arturo Ribagorda. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials*, 16(2):961–987, May 2014.

[34] Ken Dunham, Shane Hartman, Manu Quintans, Jose Andre Morales, and Tim Strazzere. *Android Malware and Analysis*. Auerbach Publications, 2014.

[35] Carlos A Castillo et al. Android malware past, present, and future. *White Paper of McAfee Mobile Security Working Group*, 1:16, 2011.

[36] Marc Fossi, Gerry Egan, Kevin Haley, Eric Johnson, Trevor Mack, Téo Adams, Joseph Blackbird, Mo King Low, Debbie Mazurek, David McKinney, et al. Symantec internet security threat report trends for 2010. *Volume XVI*, 2011.

[37] Fake apps affect android os users. https://www.trendmicro.com, (accessed July 21, 2018).

[38] Tim Wyatt. Security alert: Geinimi, sophisticated new android trojan found in wild. https://blog.mylookout.com/2010/12/geinimi_trojan/, (accessed July 21, 2018).

[39] Yajin Zhou and Xuxian Jiang. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 95–109. IEEE, 2012.

[40] Symantec internet security threat report trends for 2011. Technical report, Symantec Corporation, 2012.

[41] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14. ACM, 2011.

[42] Symantec internet security threat report trends for 2012. Technical report, Symantec Corporation, 2013.
[43] Annual security roundup. Technical report, TrendLabs, 2012.
[44] Symantec internet security threat report trends for 2013. Technical report, Symantec Corporation, 2014.
[45] Threat report h2 2012. Technical report, F-Secure, 2013.
[46] Malicious developers release rogue bad piggies versions. https://blog.trendmicro.com, (accessed July 23, 2018).
[47] Rogue instagram and angry birds space for android spotted. https://blog.trendmicro.com, (accessed July 23, 2018).
[48] 1,730 malicious apps still available on popular android app providers. https://blog.trendmicro.com, (accessed July 23, 2018).
[49] User privacy plunges as android aggressive adware and malware rise. https://www.bitdefender.com, (accessed July 23, 2018).
[50] Threat report h1 2013. Technical report, F-Secure, 2013.
[51] Annual security roundup. Technical report, TrendLabs, 2013.
[52] Threat report h2 2013. Technical report, F-Secure, 2014.
[53] Symantec internet security threat report trends for 2014. Technical report, Symantec Corporation, 2015.
[54] Threat report h1 2014. Technical report, F-Secure, 2014.
[55] Threat report h2 2014. Technical report, F-Secure, 2015.
[56] 1q 2014 security roundup. Technical report, TrendLabs, 2014.
[57] 2q 2014 security roundup. Technical report, TrendLabs, 2014.
[58] Threat report 2015. Technical report, F-Secure, 2016.
[59] Symantec internet security threat report trends for 2015. Technical report, Symantec Corporation, 2016.
[60] Annual security roundup. Technical report, TrendLabs, 2016.
[61] Annual security roundup. Technical report, TrendLabs, 2017.
[62] Symantec internet security threat report trends for 2016. Technical report, Symantec Corporation, 2017.
[63] Mobile malware evolution 2016. Technical report, Kaspersky Lab, 2017.
[64] Diwakar Dinkar, Paula Greve, Kent Landfield, François Paget, Eric Peterson, Craig Schmugar, Rakesh Sharma, Rick Simon, Bruce Snell, Dan Sommer, and Bing Sun. Mcafee labs threats report. Technical report, McAfee Labs, 2016.

[65] Wilson Cheng, Shaina Dailey, Douglas Frosst, Paula Greve, Tim Hux, Jeannette Jarvis, Abhishek Karnik, Sanchit Karve, Charles McFarland, Francisca Moreno, Igor Muttik, Mark Olea, François Paget, Ted Pan, Eric Peterson, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, and Guilherme Venere. Mcafee labs threats report. Technical report, McAfee Labs, 2016.

[66] Christiaan Beek, Joseph Fiorella, Celeste Fralick, Douglas Frosst, Paula Greve, Andrew Marwan, François Paget, Ted Pan, Eric Peterson, Craig Schmugar, Rick Simon, Dan Sommer, and Bing Sun. Mcafee labs threats report. Technical report, McAfee Labs, 2016.

[67] Christiaan Beek, Douglas Frosst, Paula Greve, Barbara Kay, Bart Lenaerts-Bergmans, Charles McFarland, Eric Peterson, Raj Samani, Craig Schmugar, Rick Simon, Dan Sommer, and Bing Sun. Mcafee labs threats report. Technical report, McAfee Labs, 2016.

[68] Christiaan Beek, Diwakar Dinkar, Yashashree Gund, German Lancioni, Niamh Minihane, Francisca Moreno, Eric Peterson, Thomas Roccia, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, RaviKant Tiwari, and Vincent Weafer. Mcafee labs threats report. Technical report, McAfee Labs, 2017.

[69] Christiaan Beek, Douglas Frosst, Paula Greve, Yashashree Gund, Francisca Moreno, Eric Peterson, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, Ravikant Tiwari, and Vincent Weafer. Mcafee labs threats report. Technical report, McAfee Labs, 2017.

[70] Christiaan Beek, Diwakar Dinkar, Douglas Frosst, Elodie Grandjean, Francisca Moreno, Eric Peterson, Prajwala Rao, Raj Samani, Craig Schmugar, Rick Simon, Dan Sommer, Bing Sun, Ismael Valenzuela, and Vincent Weafer. Mcafee labs threats report. Technical report, McAfee Labs, 2017.

[71] Niamh Minihane, Francisca Moreno, Eric Peterson, Raj Samani, Craig Schmugar, Dan Sommer, and Bing Sun. Mcafee labs threats report. Technical report, McAfee Labs, 2017.

[72] Annual security roundup. Technical report, TrendLabs, 2018.

[73] Quick heal annual threat report. Technical report, Seqrite, 2018.

[74] Kaspersky lab threat predictions for 2018. Technical report, Kaspersky Lab, 2018.

[75] Sophoslabs 2018 malware forecast. Technical report, SophosLabs, 2018.

[76] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, Konrad Rieck, and CERT Siemens. Drebin: Effective and explainable detection of android malware in your pocket. In *NDSS*, 2014.

[77] Contagio mobile malware dataset. http://contagiominidump.blogspot.com, (accessed February 19, 2018).

[78] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security*, 51:16–31, 2015.

[79] Fengguo Wei, Yuping Li, Sankardas Roy, Xinming Ou, and Wu Zhou. Deep ground truth analysis of current android malware. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA'17)*, pages 252–276, Bonn, Germany, 2017. Springer.

[80] Androzoo. https://androzoo.uni.lu/, (accessed September 10, 2018).

[81] Praguard. http://pralab.diee.unica.it/en/AndroidPRAGuardDataset, (accessed September 10, 2018).

[82] Virustotal. https://www.virustotal.com/#/home/search, (accessed September 10, 2018).

[83] Kevin Allix, Tegawendé F Bissyandé, Jacques Klein, and Yves Le Traon. Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE, 2016.

[84] F-droid. https://f-droid.org, (accessed February 10, 2018).

[85] Jd-gui. http://jd.benow.ca/, (accessed September 1, 2018).

[86] Jad. http://varaneckas.com/jad/, (accessed September 1, 2018).

[87] Dare. http://siis.cse.psu.edu/dare/source.html, (accessed September 1, 2018).

[88] Mocha. http://www.brouhaha.com/~eric/software/mocha/, (accessed September 1, 2018).

[89] Procyon. https://bitbucket.org/mstrobel/procyon, (accessed September 1, 2018).

[90] Dad. https://github.com/androguard/androguard/tree/master/androguard/decompiler/dad, (accessed September 1, 2018).

[91] Jeb. https://www.pnfsoftware.com/, (accessed September 1, 2018).

[92] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[93] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[94] Lovely Sinha, Shweta Bhandari, Parvez Faruki, Manoj Singh Gaur, Vijay Laxmi, and Mauro Conti. Flowmine: Android app analysis via data flow. In *2016 13th IEEE Annual Consumer Communications & Networking Conference (CCNC)*, pages 435–441, 2016.

[95] Long Lu, Zhichun Li, Zhenyu Wu, Wenke Lee, and Guofei Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *ACM conference on Computer and communications security*, pages 229–240, 2012.

[96] Zhemin Yang and Min Yang. Leakminer: Detect information leakage on android with static taint analysis. In *Software Engineering (WCSE)*, pages 101–104, 2012.

[97] Clint Gibler, Jonathan Crussell, Jeremy Erickson, and Hao Chen. *AndroidLeaks: automatically detecting potential privacy leaks in android applications on a large scale*. 2012.

[98] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):5, 2014.

[99] Bouncer. http://googlemobile.blogspot.com/2012/02/android-and-security.html, (accessed August 12, 2018).

[100] Vincent Haupert, Dominik Maier, Nicolas Schneider, Julian Kirsch, and Tilo Müller. Honey, i shrunk your app security: The state of android app hardening. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 69–91. Springer, 2018.

[101] Omid Mirzaei, Jose M. de Fuentes, Juan Tapiador, and Lorena Gonzalez-Manzano. Androdet: An adaptive android obfuscation detector. *Future Generation Computer Systems*, 90:240–261, 2019.

[102] Dominik Wermke, Nicolas Huaman, Yasemin Acar, Brad Reaves, Patrick Traynor, and Sascha Fahl. A large scale investigation of obfuscation use in google play. In *Proceedings of the 34th Annual Conference on Computer Security Applications*. ACM, 2018.

[103] Yuxin Gao, Zexin Lu, and Yuqing Luo. Survey on malware anti-analysis. In *Intelligent Control and Information Processing (ICICIP), 2014 Fifth International Conference on*, pages 270–275. IEEE, 2014.

[104] Vaibhav Rastogi, Yan Chen, and Xuxian Jiang. Droidchameleon: evaluating android anti-malware against transformation attacks. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 329–334. ACM, 2013.

[105] Rui Tanabe, Wataru Ueno, Kou Ishii, Katsunari Yoshioka, Tsutomu Matsumoto, Takahiro Kasama, Daisuke Inoue, and Christian Rossow. Evasive malware via identifier implanting. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 162–184. Springer, 2018.

[106] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. Rage against the virtual machine: hindering dynamic analysis of android malware. In *Proceedings of the Seventh European Workshop on System Security*, page 5. ACM, 2014.

[107] Sumeet Dua and Xian Du. *Data mining and machine learning in cybersecurity*. CRC press, 2016.

[108] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):41, 2017.

[109] Alexey Tsymbal. The problem of concept drift: definitions and related work. *Computer Science Department, Trinity College Dublin*, 106(2), 2004.

[110] Jean Paul Barddal, Heitor Murilo Gomes, Fabrício Enembreck, and Bernhard Pfahringer. A survey on feature drift adaptation: Defini-

tion, benchmark, challenges and future directions. *Journal of Systems and Software*, 127:278–294, 2017.

[111] Indrė Žliobaitė, Albert Bifet, Jesse Read, Bernhard Pfahringer, and Geoff Holmes. Evaluation methods and decision theory for classification of streaming data with temporal dependence. *Machine Learning*, 98(3):455–482, 2015.

[112] Heitor Murilo Gomes, Jean Paul Barddal, Fabrício Enembreck, and Albert Bifet. A survey on ensemble learning for data stream classification. *ACM Computing Surveys (CSUR)*, 50(2):23, 2017.

[113] Albert Bifet, Geoff Holmes, Richard Kirkby, and Bernhard Pfahringer. Moa: Massive online analysis. *Journal of Machine Learning Research*, 11(May):1601–1604, 2010.

[114] Gianmarco De Francisci Morales and Albert Bifet. Samoa: scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16(1):149–153, 2015.

[115] Peter Reutemann and Joaquin Vanschoren. Scientific workflow management with adams. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 833–837. Springer, 2012.

[116] Shohei Hido, Seiya Tokui, and Satoshi Oda. Jubatus: An open source platform for distributed online machine learning. In *NIPS 2013 Workshop on Big Learning, Lake Tahoe*, 2013.

[117] Vowpal. https://github.com/JohnLangford/vowpal_wabbit, (accessed February 12, 2018).

[118] Streamdm. http://huawei-noah.github.io/streamDM, (accessed February 12, 2018).

[119] Muhammad Habib ur Rehman, Chee Sun Liew, and Teh Ying Wah. Frequent pattern mining in mobile devices: A feasibility study. In *information technology and multimedia (ICIMU), 2014 International Conference on*, pages 351–356. IEEE, 2014.

[120] Hong Cao and Miao Lin. Mining smartphone data for app usage prediction and recommendations: A survey. *Pervasive and Mobile Computing*, 2017.

[121] Samuel Huppe, Mohamed Aymen Saied, and Houari Sahraoui. Mining complex temporal api usage patterns: an evolutionary approach. In *Proceedings of the 39th International Conference on Software Engineering Companion*, pages 274–276. IEEE Press, 2017.

[122] Kuo-Wei Hsu. Effectively mining time-constrained sequential patterns of smartphone application usage. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, page 39. ACM, 2017.

[123] Md Yasser Karim, Huzefa Kagdi, and Massimiliano Di Penta. Mining android apps to recommend permissions. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 427–437. IEEE, 2016.

[124] Xinli Yang, David Lo, Li Li, Xin Xia, Tegawendé F Bissyandé, and Jacques Klein. Characterizing malicious android apps by mining topic-specific data flow signatures. *Information and Software Technology*, 2017.

[125] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[126] Chengqi Zhang and Shichao Zhang. *Association rule mining: models and algorithms*. Springer-Verlag, 2002.

[127] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2009.

[128] Borja Sanz, Igor Santos, Carlos Laorden, Xabier Ugarte-Pedrero, Pablo Garcia Bringas, and Gonzalo Álvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS'12-ICEUTE 12-SOCO 12 Special Sessions*, pages 289–298. Springer, 2013.

[129] Yousra Aafer, Wenliang Du, and Heng Yin. Droidapiminer: Mining api-level features for robust malware detection in android. In *International conference on security and privacy in communication systems*, pages 86–103. Springer, 2013.

[130] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, pages 45–54. ACM, 2013.

[131] Yu Feng, Saswat Anand, Isil Dillig, and Alex Aiken. Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM, 2014.

[132] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, pages 15–26. ACM, 2011.

[133] Yajin Zhou, Zhi Wang, Wu Zhou, and Xuxian Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS*, 2012.

[134] Min Zheng, Mingshen Sun, and John Lui. Droidanalytics: a signature based analytic system to collect, extract, analyze and associate android malware. *arXiv preprint arXiv:1302.7212*, 2013.

[135] Michael Spreitzenbarth, Felix Freiling, Florian Echtler, Thomas Schreck, and Johannes Hoffmann. Mobile-sandbox: having a deeper look into android applications. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1808–1815. ACM, 2013.

[136] Rakesh Agrawal, Ramakrishnan Srikant, et al. Fast algorithms for mining association rules. In *Proc. 20th int. conf. very large data bases, VLDB*, volume 1215, pages 487–499, 1994.

[137] Jiawei Han, Jian Pei, Yiwen Yin, and Runying Mao. Mining frequent patterns without candidate generation: A frequent-pattern tree approach. *Data mining and knowledge discovery*, 8(1):53–87, 2004.

[138] Ling Huang, Anthony D Joseph, Blaine Nelson, Benjamin IP Rubinstein, and JD Tygar. Adversarial machine learning. In *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, pages 43–58. ACM, 2011.

[139] Ambra Demontis, Marco Melis, Battista Biggio, Davide Maiorca, Daniel Arp, Konrad Rieck, Igino Corona, Giorgio Giacinto, and Fabio Roli. Yes, machine learning can be more secure! a case study on android malware detection. *IEEE Transactions on Dependable and Secure Computing*, 2017.

[140] Marco Melis, Davide Maiorca, Battista Biggio, Giorgio Giacinto, and Fabio Roli. Explaining black-box android malware detection. *arXiv preprint arXiv:1803.03544*, 2018.

[141] Pavel Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[142] Weilin Xu, Yanjun Qi, and David Evans. Automatically evading classifiers. In *Proceedings of the 2016 Network and Distributed Systems Symposium*, 2016.

[143] Zainab Abaid, Mohamed Ali Kaafar, and Sanjay Jha. Quantifying the impact of adversarial evasion attacks on machine learning based android malware classifiers. In *Network Computing and Applications (NCA), 2017 IEEE 16th International Symposium on*, pages 1–10. IEEE, 2017.

[144] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 288–302. ACM, 2017.

[145] Xiao Chen, Chaoran Li, Derui Wang, Sheng Wen, Jun Zhang, Surya Nepal, Yang Xiang, and Kui Ren. Android hiv: A study of repackaging malware for evading machine-learning detection. *arXiv preprint arXiv:1808.04218*, 2018.

[146] Alejandro Calleja, Alejandro Martín, Héctor D Menéndez, Juan Tapiador, and David Clark. Picking on the family: Disrupting android malware triage by forcing misclassification. *Expert Systems with Applications*, 95:113–126, 2018.

[147] Lingwei Chen, Shifu Hou, Yanfang Ye, and Shouhuai Xu. Droideye: Fortifying security of learning-based classifier against adversarial android malware attacks. In *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pages 782–789. IEEE, 2018.

[148] Threats report. Technical report, McAfee, 2016.

[149] Enrico Mariconti, Lucky Onwuzurike, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini. Mamadroid: Detecting android malware by building markov chains of behavioral models. In *The Network and Distributed System Security Symposium (NDSS)*, 2017.

[150] William Enck, Machigar Ongtang, and Patrick McDaniel. On lightweight mobile phone application certification. In *16th ACM Conference on Computer and Communications Security*, CCS '09, pages 235–245, 2009.

[151] Michael Grace, Yajin Zhou, Qiang Zhang, Shihong Zou, and Xuxian Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, pages 281–294. ACM, 2012.

[152] Yiming Jing, Gail-Joon Ahn, Ziming Zhao, and Hongxin Hu. Riskmon: Continuous and automated risk assessment of mobile applications. In *4th ACM Conference on Data and Application Security and Privacy*, CODASPY '14, pages 99–110, 2014.

[153] Hao Peng, Chris Gates, Bhaskar Sarma, Ninghui Li, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Using probabilistic generative models for ranking risks of android apps. In *ACM Conference on Computer and Communications Security*, CCS '12, pages 241–252, 2012.

[154] Bhaskar Pratim Sarma, Ninghui Li, Chris Gates, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Android permissions: A perspective combining risks and benefits. In *17th ACM Symposium on Access Control Models and Technologies*, SACMAT '12, pages 13–22, 2012.

[155] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick Drew McDaniel, and Matthew Smith. Sok: Lessons learned from android security research for appified software platforms. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 433–451, 2016.

[156] Daniel Arp, Michael Spreitzenbarth, Malte Hubner, Hugo Gascon, and Konrad Rieck. Drebin: Effective and explainable detection of Android malware in your pocket. In *Network and Distributed System Security Symposium (NDSS)*. 2014.

[157] Vitalii Avdiienko, Konstantin Kuznetsov, Alessandra Gorla, Andreas Zeller, Steven Arzt, Siegfried Rasthofer, and Eric Bodden. Mining apps for abnormal usage of sensitive data. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 426–436. IEEE Press, 2015.

[158] Christopher S. Gates, Ninghui Li, Hao Peng, Bhaskar Pratim Sarma, Yuan Qi, Rahul Potharaju, Cristina Nita-Rotaru, and Ian Molloy. Generating summary risk scores for mobile applications. *IEEE Trans. Dependable Sec. Comput.*, 11(3):238–251, 2014.

[159] Siegfried Rasthofer, Steven Arzt, and Eric Bodden. A machine-learning approach for classifying and categorizing android sources and sinks. In *2014 Network and Distributed System Security Symposium (NDSS)*, 2014.

[160] Wei Chen, David Aspinall, Andrew D. Gordon, Charles Sutton, and Igor Muttik. More semantics more robust: Improving android malware classifiers. In *9th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '16, pages 147–158, 2016.

[161] Sankardas Roy, Jordan DeLoach, Yuping Li, Nic Herndon, Doina Caragea, Xinming Ou, Venkatesh Prasad Ranganath, Hongmin Li, and Nicolais Guevara. Experimental study with real-world data for android app security analysis using machine learning. In *31st Annual Computer Security Applications Conference*, pages 81–90, 2015.

[162] Yang Wang, Jun Zheng, Chen Sun, and Srinivas Mukkamala. Quantitative security risk assessment of android permissions and applications. In *Data and Applications Security and Privacy*, pages 226–241, 2013.

[163] Benjamin Andow, Adwait Nadkarni, Blake Bassett, William Enck, and Tao Xie. A study of grayware on google play. In *2016 IEEE Security and Privacy Workshops, SP Workshops 2016, San Jose, CA, USA, May 22-26, 2016*, pages 224–233, 2016.

[164] Adrienne Porter Felt, Helen J. Wang, Alexander Moshchuk, Steve Hanna, and Erika Chin. Permission re-delegation: Attacks and defenses. In *USENIX Security Symposium*, 2011.

[165] Guillermo Suarez-Tangil, Juan E. Tapiador, and Pedro Peris-Lopez. Compartmentation policies for android apps: A combinatorial optimization approach. In *Int. Conf. Network and System Security (NSS)*, pages 63–77, 2015.

[166] Li Li, Alexandre Bartel, Tegawendé F. Bissyandé, Jacques Klein, and Yves Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *ICT Systems Security and Privacy Protection: 30th IFIP TC 11 International Conference, SEC 2015, Hamburg, Germany, May 26-28, 2015, Proceedings*, pages 513–527, 2015.

[167] Wei You, Bin Liang, Jingzhe Li, Wenchang Shi, and Xiangyu Zhang. Android implicit information flow demystified. In *10th*

ACM Symposium on Information, Computer and Communications Security, pages 585–590, 2015.

[168] Man-Ki Yoon, Negin Salajegheh, Yin Chen, and Mihai Christodorescu. Pift: Predictive information-flow tracking. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 713–725, 2016.

[169] Jae-wook Jang, Hyunjae Kang, Jiyoung Woo, Aziz Mohaisen, and Huy Kang Kim. Andro-dumpsys: anti-malware system based on the similarity of malware creator and malware centric information. *computers & security*, 58:125–138, 2016.

[170] Ali Razeen, Valentin Pistol, Alexander Meijer, and Landon P Cox. Better performance through thread-local emulation. In *17th International Workshop on Mobile Computing Systems and Applications*, pages 87–92, 2016.

[171] Parvez Faruki, Shweta Bhandari, Vijay Laxmi, Manoj Gaur, and Mauro Conti. Droidanalyst: Synergic app framework for static and dynamic app analysis. In *Recent Advances in Computational Intelligence in Defense and Security*, pages 519–552. 2016.

[172] Erika Chin, Adrienne Porter Felt, Kate Greenwood, and David Wagner. Analyzing inter-application communication in android. In *9th international conference on Mobile systems, applications, and services*, pages 239–252, 2011.

[173] Roei Schuster and Eran Tromer. Droiddisintegrator: Intra-application information flow control in android apps. In *11th ACM Asia Conference on Computer and Communications Security*, 2016.

[174] William Klieber, Lori Flynn, Amar Bhosale, Limin Jia, and Lujo Bauer. Android taint flow analysis for app sets. In *3rd ACM SIGPLAN International Workshop on the State of the Art in Java Program Analysis*, pages 1–6, 2014.

[175] Wei Huang, Yao Dong, Ana Milanova, and Julian Dolby. Scalable and precise taint analysis for android. In *International Symposium on Software Testing and Analysis*, pages 106–117, 2015.

[176] Guillermo Suarez-Tangil, Juan E Tapiador, and Pedro Peris-Lopez. Stegomalware: Playing hide and seek with malicious components in smartphone apps. In *10th International Conference on Information Security and Cryptology (Inscrypt)*, pages 496–515, 2014.

[177] Saurabh Chakradeo, Bradley Reaves, Patrick Traynor, and William Enck. Mast: Triage for market-scale mobile malware analysis. In *Sixth ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '13, pages 13–24, 2013.

[178] Rahul Pandita, Xusheng Xiao, Wei Yang, William Enck, and Tao Xie. Whyper: Towards automating risk assessment of mobile applications. In *USENIX Security*, volume 13, 2013.

[179] Andrea Saracino, Daniele Sgandurra, Gianluca Dini, and Fabio Martinelli. Madam: Effective and efficient behavior-based android malware detection and prevention. *IEEE Transactions on Dependable and Secure Computing*, (99), 2016.

[180] Alexios Mylonas, Marianthi Theoharidou, and Dimitris Gritzalis. Assessing privacy risks in android: A user-centric approach. In *Risk Assessment and Risk-Driven Testing*, pages 21–37. 2013.

[181] Shancang Li, Theo Tryfonas, Gordon Russell, and Panagiotis Andriotis. Risk assessment for mobile systems through a multilayered hierarchical bayesian network. *IEEE Transactions on Cybernetics*, (99):1–11, 2016.

[182] Security report 2016/17. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2016-2017.pdf, (accessed May 6, 2018).

[183] Mobile malware evolution. https://securelist.com/mobile-malware-review-2017/84139/, (accessed May 6, 2018).

[184] Sevil Sen, Emre Aydogan, and Ahmet I Aysan. Coevolution of mobile malware and anti-malware. *IEEE Transactions on Information Forensics and Security*, 2018.

[185] Omid Mirzaei, Guillermo Suarez-Tangil, Juan Tapiador, and Jose M de Fuentes. Triflow: Triaging android applications using speculative information flows. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 640–651. ACM, 2017.

[186] Gosta Grahne and Jianfei Zhu. Fast algorithms for frequent itemset mining using fp-trees. *IEEE Trans. on Knowl. and Data Eng.*, 17(10):1347–1362, October 2005.

[187] Philippe Fournier-Viger, Jerry Chun-Wei Lin, Antonio Gomariz, Ted Gueniche, Azadeh Soltani, Zhihong Deng, and Hoang Thanh

Lam. *The SPMF Open-Source Data Mining Library Version 2*, pages 36–40. Springer International Publishing, Cham, 2016.

[188] Wei Chen, David Aspinall, Andrew D Gordon, Charles Sutton, and Igor Muttik. A text-mining approach to explain unwanted behaviours. In *Proceedings of the 9th European Workshop on System Security*, page 4. ACM, 2016.

[189] Joshua Garcia, Mahmoud Hammad, and Sam Malek. Lightweight, obfuscation-resilient detection and family identification of android malware. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):11, 2018.

[190] Anil K Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[191] Guojun Gan and Michael Kwok-Po Ng. k-means clustering with outlier removal. *Pattern Recognition Letters*, 90:8–14, 2017.

[192] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics, 2007.

[193] Peter J Rousseeuw. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20:53–65, 1987.

[194] Jiawei Han, Jian Pei, and Micheline Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[195] Yuping Li, Jiyong Jang, Xin Hu, and Xinming Ou. Android malware clustering through malicious payload mining. In *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 192–214. Springer, 2017.

[196] Parvez Faruki, Ammar Bharmal, Vijay Laxmi, Vijay Ganmoor, Manoj Singh Gaur, Mauro Conti, and Muttukrishnan Rajarajan. Android security: a survey of issues, malware penetration, and defenses. *IEEE communications surveys & tutorials*, 17(2):998–1022, 2015.

[197] Sebastian Zimmeck, Ziqi Wang, Lieyong Zou, Roger Iyengar, Bin Liu, Florian Schaub, Shomir Wilson, Norman Sadeh, Steven Bellovin, and Joel Reidenberg. Automated analysis of privacy requirements for mobile apps. In *2016 AAAI Fall Symposium Series*, 2016.

[198] Jiang Ming, Dinghao Wu, Gaoyao Xiao, Jun Wang, and Peng Liu. Taintpipe: Pipelined symbolic taint analysis. In *USENIX Security*, volume 15, 2015.

[199] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. Supor: Precise and scalable sensitive user input detection for android apps. In *USENIX Security*, volume 15, 2015.

[200] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 377–396. IEEE, 2016.

[201] Guillermo Suarez-Tangil, Santanu Kumar Dash, Mansour Ahmadi, Johannes Kinder, Giorgio Giacinto, and Lorenzo Cavallaro. Droidsieve: Fast and accurate classification of obfuscated android malware. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 309–320. ACM, 2017.

[202] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 2012 Seventh Asia Joint Conference on Information Security*, ASIAJCIS '12, pages 62–69, Washington, DC, USA, 2012. IEEE Computer Society.

[203] Marius Gheorghescu. An automated virus classification system. In *Virus bulletin conference*, volume 2005, pages 294–300. Citeseer, 2005.

[204] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[205] Joris Kinable and Orestis Kostakis. Malware classification based on call graph clustering. *Journal in computer virology*, 7(4):233–245, 2011.

[206] Guillermo Suarez-Tangil and Gianluca Stringhini. Eight years of rider measurement in the android malware ecosystem: Evolution and lessons learned. *arXiv preprint arXiv:1801.08115*, 2018.

[207] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. *Department of Computer Science, George Mason University, Tech. Rep*, 2015.

[208] Ming Fan, Jun Liu, Xiapu Luo, Kai Chen, Zhenzhou Tian, Qinghua Zheng, and Ting Liu. Android malware familial classification and representative sample selection via frequent subgraph analysis. *IEEE Transactions on Information Forensics and Security*, 2018.

[209] Androsim. https://github.com/vivainio/androguard/blob/master/androsim.py, (accessed June 1, 2018).

[210] Dexid. https://github.com/bontchev/dexid, (accessed June 1, 2018).

[211] Shun-Wen Hsiao, Yeali S Sun, and Meng Chang Chen. Behavior grouping of android malware family. In *Communications (ICC), 2016 IEEE International Conference on*, pages 1–6. IEEE, 2016.

[212] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[213] Antonio Bianchi, Yanick Fratantonio, Aravind Machiry, Christopher Kruegel, Giovanni Vigna, Simon Pak Ho Chung, and Wenke Lee. Broken fingers: On the usage of the fingerprint api in android. In *NDSS'18*, 2018.

[214] Smartphone os market share. https://www.idc.com/promo/smartphone-market-share/os, (accessed February 19, 2018).

[215] Mobile malware evolution. https://securelist.com/mobile-malware-review-2017/84139/, (accessed March 14, 2018).

[216] Albert Bifet and Richard Kirkby. Data stream mining a practical approach. 2009.

[217] Thomas Swearingen, Will Drevo, Bennett Cyphers, Alfredo Cuesta-infante, Arun Ross, and Kalyan Veeramachaneni. ATM : A distributed , collaborative , scalable system for automated machine learning. (December), 2017.

[218] Allatori. http://www.allatori.com/, (accessed February 10, 2018).

[219] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[220] Sebastian Banescu and Alexander Pretschner. A tutorial on software obfuscation. *Advances in Computers. Elsevier*, 2018.

[221] Vivek Balachandran, Darell JJ Tan, Vrizlynn LL Thing, et al. Control flow obfuscation for android applications. *Computers & Security*, 61:72–93, 2016.

[222] Juanru Li, Dawu Gu, and Yuhao Luo. Android malware forensics: Reconstruction of malicious events. In *Distributed Computing Systems Workshops (ICDCSW), 2012 32nd International Conference on*, pages 552–558. IEEE, 2012.

[223] Dasho. https://www.preemptive.com/products/dasho/overview, (accessed February 10, 2018).

[224] Nak Young Kim, Jaewoo Shim, Seong-je Cho, Minkyu Park, and Sangchul Han. Android application protection against static reverse engineering based on multidexing. *J. Internet Serv. Inf. Secur.*, 6(4):54–64, 2016.

[225] Hyunwoo Choi and Yongdae Kim. Large-scale analysis of remote code injection attacks in android apps. *Security and Communication Networks*, 2018, 2018.

[226] Tree-based feature selection. http://scikit-learn.org/stable/modules/feature_selection.html, (accessed March 17, 2018).

[227] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar, and Abdullah Gani. Evaluation of machine learning classifiers for mobile malware detection. *Soft Computing*, 20(1):343–357, 2016.

[228] James S Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *Advances in neural information processing systems*, pages 2546–2554, 2011.

[229] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[230] Nick Littlestone and Manfred K Warmuth. The weighted majority algorithm. *Information and computation*, 108(2):212–261, 1994.

[231] Albert Bifet, Geoff Holmes, and Bernhard Pfahringer. Leveraging bagging for evolving data streams. In *Joint European conference on machine learning and knowledge discovery in databases*, pages 135–150. Springer, 2010.

[232] Meenakshi A Thalor and ST Patil. Ensemble for non stationary data stream: Performance improvement over learn++. NSE. In *Information Processing (ICIP), 2015 International Conference on*, pages 225–228. IEEE, 2015.

[233] Christophe Salperwyck, Vincent Lemaire, and Carine Hue. Incremental weighted naive bays classifiers for data stream. In *Data Science, Learning by Latent Structures, and Knowledge Discovery*, pages 179–190. Springer, 2015.

[234] Ingo Steinwart and Andreas Christmann. *Support vector machines*. Springer Science & Business Media, 2008.

[235] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[236] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[237] Evaluateprequential. https://www.cs.waikato.ac.nz/~abifet/MOA/API/classmoa_1_1tasks_1_1_evaluate_prequential.html, (accessed March 12, 2018).

[238] Mila Dalla Preda and Federico Maggi. Testing android malware detectors against code obfuscation: a systematization of knowledge and unified methodology. *Journal of Computer Virology and Hacking Techniques*, 13(3):209–232, 2017.

[239] Joshua Garcia, Mahmoud Hammad, Bahman Pedrood, Ali Bagheri-Khaligh, and Sam Malek. Obfuscation-resilient, efficient, and accurate detection and family identification of android malware. Technical report, Dept. of Computer Science, George Mason University, 2015.

[240] Fangfang Zhang, Heqing Huang, Sencun Zhu, Dinghao Wu, and Peng Liu. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 25–36. ACM, 2014.

[241] Leonid Glanz, Sven Amann, Michael Eichberg, Michael Reif, Ben Hermann, Johannes Lerch, and Mira Mezini. Codematch: obfuscation won't conceal your repackaged app. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 638–648. ACM, 2017.

[242] Richard Baumann, Mykolai Protsenko, and Tilo Müller. Antiproguard: Towards automated deobfuscation of android apps. In *Proceedings of the 4th Workshop on Security in Highly Connected IT Systems*, pages 7–12. ACM, 2017.

[243] Woojong Yoo, Myeongju Ji, Minkoo Kang, and Jeong Hyun Yi. String Deobfuscation Scheme based on Dynamic Code Extraction for Mobile Malwares. 2(June):1–8, 2016.

[244] Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin Vechev. Statistical deobfuscation of android applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 343–355. ACM, 2016.